RL-TR-97-20
Final Technical Report
June 1997

CERTIFICATION OF REUSABLE SOFTWARE COMPONENTS

Software Productivity Solutions, Inc.

Sharon Rohde and Pamela Geriner

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

DTIC QUALITY INSPECTED 4

19970728 130

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-20 has been reviewed and is approved for publication.

APPROVED: Deborah A. Cerino

DEBORAH A. CERINO
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO, Chief Scientist
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3CB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1997	Final Dec 93 - Dec 96

4. TITLE AND SUBTITLE

CERTIFICATION OF REUSABLE SOFTWARE COMPONENTS

5. FUNDING NUMBERS

C - F30602-94-C-0024
PE - 62702F
PR - 5581
TA - 18
WU - 61

6. AUTHOR(S)

Sharon Rohde and Pamela Geriner

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Productivity Solutions, Inc.
122 Fourth Avenue
Indialantic FL 32903

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/C3CB
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-20

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Deborah A. Cerino/C3CB/(315) 330-2054

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Recognizing that software will not be reused unless its quality can be accurately and effectively determined, Rome Laboratory has initiated a research project in reusable software asset certification. Certification, as used in this document, refers to a process in which inspection, analysis, and testing techniques are used to achieve assurance of the quality of reusable assets. This process might be performed by a reuse repository, by a reuser, by an independent organization providing such services, or by a development organization. This effort provides a certification framework that defines methods and tools that can be applied to detect defects and ultimately avoid rework. The framework focuses on certifying individual components (i.e., smaller pieces of a system) for a particular quality or group of qualities. These individual components can then be used as system building blocks.

14. SUBJECT TERMS

Software Certification, Software Assessment and Evaluation

15. NUMBER OF PAGES

264

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

Table of Contents

List of Figures

List of Tables

Contributors to the CRC Project

Listed in alphabetical order, the following persons contributed to the CEC Project:

Lynda L. Burns, Software Productivity Solutions, Inc.

David N. Card, Software Productivity Solutions, Inc.

Deborah A. Cerino, Rome Laboratory of the U.S. Air Force Material Command

Edward R. Comer, InQuisiX, Inc.

Karen A. Dyson, Software Productivity Solutions, Inc.

Janet Flynt, Underweiters Laboratories, Inc.

Pamela T. Geriner, Ph.D., Software Productivity Solutions, Inc.

Jeffrey A. Heimberger, Software Productivity Solutions, Inc.

Duane W. Hybertson, Ph.D., The MITRE Corporation

Holly G. Mills, Software Productivity Solutions, Inc.

Sharon L. Rhode Software Productivity Solutions, Inc.

Charlotte O. Scheper, VeriQuest, LLC

Sharon Smith, Underwriters Laboratories, Inc.

Tom Strelich, General Research Corporation

William M. Thomas, The MITRE Corporation

Executive Summary

It has been estimated that the U.S. Department of Defense (DoD) spends in excess of $24 billion per year to develop and maintain software for weapons, command and control, and other automated information systems [GAO93]. The increase in number and size of software intensive systems has led to rising software development and maintenance costs. Consequently, the DoD needs to identify methods that will accelerate development schedules, lower cost, and improve quality.

Software component reuse and certification are two technologies that have great potential to counteract the rising costs of software development and maintenance. *Certification*, as defined in this and related documents, refers to a process by which inspection, analysis, and testing techniques are used to achieve assurance of the quality of reusable assets. Certification is expected to stimulate component reuse and reduce the amount of rework required [DUN92]. The certification process is performed by a reuse repository, by a reuser, by an independent organization providing such services, or by a development organization.

As more and more organizations embark on software reuse programs, the need for a comprehensive and systematic approach to component reuse and certification becomes essential. Organizations need guidance within their reuse programs to assess the benefits of certification in terms of risk reduction and cost savings. Recognizing that software will not be reused unless its quality can be accurately and effectively determined, Rome Laboratory (RL) of the United States Air Force Materiel Command established a research program in reusable software asset certification. The goal of this technology thrust at RL was to make certification usable, practical, and cost-effective.

In January of 1994, RL began a thirty-month, exploratory development project entitled "Certification of Reusable Software Components" (CRC). The prime contractor for CRC was Software Productivity Solutions, Inc., with subcontractors from General Research Corporation and VeriQuest, LLC.

Under the CRC contract, a Certification Framework (CF) for software components was developed which is sensitive to varying domains, business strategies and asset types. A cost benefit plan, an operational concept, and a suite of certification tools were defined. An automated prototype of the CF was developed and can be accessed on the World Wide Web through a CRC home page. A data collection guide and procedures for a certification field trial were developed, an initial field trial was conducted, and the results were analyzed and reported. Additional certification field trials are planned under separately funded contracts.

This document, the Final Technical Report (FTR), describes the work performed and the results of the CRC project. Additional supporting information is found in the following succeeding volumes of the project documentation suite:

- Volume 1 - Project Summary - summarizes the project, the reuse context and its impacts to the development of a certification framework.
- Volume 2 - Certification Framework (CF) - describes the research conducted to develop the CF.
- Volume 3 - Cost/Benefit Plan - describes a systematic approach to evaluating the costs and benefits of applying certification technology in the context of a reuse program.
- Volume 4 - Operational Concept Document (OCD) - defines the operational concept of an automated certification environment and reports the results of field interviews with potential users.
- Volume 5 - Certification Field Trial - details the procedures, collection forms, results, and lessons learned from the initial certification field trial performed by Software Productivity Solutions, Inc.
- Volume 6 - Certification Toolset - identifies the requirements for certification tools and reports the evaluation and selection of tools based on these requirements.
- Volume 7 - Code Defect Model - provides a model of code defects based on empirical data collected from studies of industry projects.
- Automated Certification Environment (ACE) System/Segment Specification (SSS) - specifies the requirements for the ACE.

The details of the work completed in each of these topic areas can be found in the designated supporting document.

1 Introduction

This document, the Final Technical Report (FTR), captures the work done under Certification of Reusable Software Components (CRC), Contract Number F30602-94-C-0024, funded by the Rome Laboratory of the Air Force Materiel Command, Rome, NY. The FTR is organized into the following sections:

- Section 1, Introduction - describes the organization of this document and the other associated volumes of project documentation.
- Section 2, Motivations - discusses the climate and incentives of reuse and certification within the software industry.
- Section 3, Definition of Certification - defines and differentiates certification for reuse.
- Section 4, The CRC Project - identifies the project goals, the CRC Team, the measures of success, our technical approach, task areas, roles and work products.
- Section 5, Reuse and Certification Technologies - discusses the history, state of the art, the state of the practice, and trends of reuse and certification.
- Section 6, Reuse Context - defines the reuse context for asset quality certification and discusses its elements.
- Section 7, Impacts of the Reuse Context to the Certification Framework - discusses the impacts of the elements of the reuse context to the development of the CF.
- Section 8, Project Results - indicates the history of the project and reports the results of the development of the Certification Framework, the Cost/Benefit Plan, the Operational Context Document, the Certification Field Trial, the Certification Toolset, and the Code Defect Model.
- Section 9, Lessons Learned - captures the experiences gained through the activities of the project.
- Section 10, Conclusions - assesses the accomplishments of the project using established measures of success and identifies implications for future areas of research.
- References - lists the references for the body of the report and its appendices.
- Appendix A, Business Strategies - the first of a subdivided annotated bibliography of a literature survey of prior research in reuse business strategies.

- Appendix B, Domain Analysis - the second of a subdivided annotated bibliography of a literature survey of prior research in domain analysis.
- Appendix C, Asset Production - the third of a subdivided annotated bibliography of a literature survey of prior research in asset production.
- Appendix D, Asset Selection - the fourth of a subdivided annotated bibliography of a literature survey of prior research in asset selection.
- Appendix E, Reuse Frameworks - the fifth of a subdivided annotated bibliography of a literature survey of prior research in reuse frameworks.
- Appendix F, Technical Paper - submitted for juried review to the Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96).

Some of the annotated bibliographies in Appendix A-E are lengthy, whereas others are very short. For those that are short, a full reference pointing to the original source appears if the reader desires further study. For those references whose annotations are lengthy, a more elaborate discussion was included since these particular references critically impacted and closely fed into the development of the Certification Framework.

Additional supporting information about the work performed under CRC is found in the following succeeding volumes of the project documentation suite:

- Volume 1 - Project Summary - summarizes the project and the reuse context and its impacts to the development of a certification framework.
- Volume 2 - Certification Framework (CF) - describes the research conducted to develop the CF.
- Volume 3 - Cost/Benefit Plan - describes a systematic approach to evaluating the costs and benefits of applying certification technology in the context of a reuse program.
- Volume 4 - Operational Concept Document (OCD) - defines the operational concept of an automated certification environment and reports the results of field interviews with potential users.
- Volume 5 - Certification Field Trial - details the procedures, collection forms, results, and lessons learned from the initial certification field trial performed by Software Productivity Solutions, Inc.
- Volume 6 - Certification Toolset - identifies the requirements for certification tools and reports the evaluation and selection of tools based on these requirements.

- Volume 7 - Code Defect Model - provides a model of code defects based on empirical data collected from studies of industry projects.

- Automated Certification Environment (ACE) System/Segment Specification (SSS) - specifies the requirements for the ACE.

The details of the work completed in each of these topic areas can be found in the designated supporting document.

2 Motivations

It has been estimated that the Department of Defense (DoD) spends in excess of $24 billion per year to develop and maintain software for weapons, command and control, and other automated information systems [GAO93]. The increase of software intensive systems in conjunction with rising software development and maintenance costs has resulted in the need to identify methods that will accelerate development schedules, lower cost, and improve quality. To address this problem, the DoD established a program, on November 25, 1991, for implementing software and other information technology initiatives as a potential solution to the upwardly spiraling costs of producing and maintaining software. As a part of this program, the Director for Defense Information proposed a software reuse initiative to build partnerships among users and suppliers of reusable components as well as the research and development community. The following ten key thrusts of this software reuse strategy are discussed in [GAO93]:

1. Specify the domains where reuse opportunities exist and identify criteria to prioritize, qualify, and select domains for application of reuse techniques.

2. Define the types of products suitable for reuse and develop criteria to validate these components for new applications.

3. Determine what ownership criteria pertain to these components and require conscious decisions regarding their ownership.

4. Modify the current acquisition process so reuse is integrated into each phase of the acquisition process and into the overall system/software life cycle.

5. Define models that may suggest novel strategies and require tailored acquisition approaches to support reuse, in order to guide business decisions.

6. Establish procedures to collect metrics that (1) measure the payoff from the reuse initiative and (2) aid developers in the selection of reusable components.

7. Define standards for the various types of components that will permit their certification for reuse.

8. Pursue a technology-based investment strategy that identifies, tracks, and transitions appropriate reuse-oriented process and product technologies.

9. Conduct comprehensive training to ensure that practitioners and policy makers capitalize on the initiative.

10. Exploit near-term products and services that facilitate movement to a reuse-based paradigm.

The Defense's software technology strategy also states that the savings from reusing software assets is estimated to be $11.3 billion in constant 1992 dollars by the year 2008. In addition, other Defense sources report that "benefits go beyond cost savings to include substantial increases in productivity for avoidance of rework, and added software quality through the use of tested components [GAO93]."

Recognizing that software will not be reused unless its quality can be accurately and effectively determined, the U.S. Air Force Rome Laboratory established a research program in reusable software asset certification. The Certification of Reusable Components (CRC) Program is one of many projects being executed within the certification initiative. RL's 20-year legacy of research in software quality, measurement, test and validation provides an excellent foundation for certification research and development. Certification is expected to stimulate component reuse and reduce the amount of rework required [DUN92].

3 Definition of Certification

The term *certification* has been used traditionally to refer to a process whereby an independent organization confirms that products meet certain requirements [ANS94]. Within the software reuse community, the term refers to a variety of activities including inspection and documentation of reusable assets as well as quality evaluation and assessment. *Certification*, as used in this and related documents, refers to a process in which inspection, analysis, and testing techniques are used to achieve assurance of the quality of reusable assets. This process might be performed by a reuse repository, by a reuser, by an independent organization providing such services, or by a development organization.

It is important at the onset, that we distinguish between the traditional process of certifying software for safety critical systems (i.e., flight and avionics systems, nuclear power systems, etc.) and the type of certification around which we are building our work products on the CRC project. Rather than confirming that a product meets certain requirements, CRC provides a certification framework, given a business strategy, a domain, an asset type, and a quality factor, that defines methods and tools that can be applied to detect defects and ultimately avoid rework. Prior to CRC, very little work had been done in certifying software from this perspective of quality and avoidance of rework.

Likewise, very little work had been done in certification of software for safety critical systems. To date, the industry has relied on hardware, not software, to achieve certifications of safety critical software. Just recently, in 1993, the U.S. Nuclear Regulatory Commission, produced a draft document that initially defines software certification and begins to address "retrofitting" safety critical nuclear plants with software systems [NRC93]. The NRC does not certify software itself; rather, a nuclear power plant is licensed. The plant's licensing includes the software that it contains. As an aside, the nuclear commissions in foreign countries (e.g., in Europe and Canada) are seeking formal methods for software assessment.

The MITRE Corporation prepared another document for the Nuclear Regulatory Commission that provides guidelines for high integrity software [MIT95]. This document examines the technical basis for candidate guidelines that could be considered in reviewing and evaluating high integrity computer software used in the safety systems of nuclear power plants. It describes approximately 200 candidate guidelines that span the entire range of software life-cycle activities; the assessment of the technical basis for those candidate guidelines; and the identification, categorization and prioritization of research needs for improving the technical basis.

With these different uses of certification in mind, it is important to distinguish between certification in general, software certification as discussed above, and software certification for reuse. CRC focuses on the later, in that we are providing a framework to "certify" individual components (i.e., smaller pieces of a system) for a particular

quality, or a group of qualities. These individual components can then be used as building blocks to devise another system, that is, a system that is composed of reusable components. The success of systems based upon reusable components may be, therefore, tied to the quality of its individual components.

Consequently, the application of reusable software is dependent upon developing an effective, systematic approach to component certification. Devising a framework that can "certify" a software component for reuse encourages component usage and can increase the quality of the overall delivered system. Certification also helps to define criteria that will determine which software components are suitable for reuse. Certification information helps determine which components to reuse and when to reuse them. While certification does not guarantee that the reused components will work as intended by the user, it does suggest the level of difficulty likely to be encountered and the probability of success of the reuse.

4 The CRC Project

The following sections identify the CRC project team, discuss the project goals and measures of success, the technical approach, task areas and roles.

4.1 Project Team

CRC began in December 1993 and was staffed with a well-qualified and experienced team throughout the life of the project until its completion in June 1996. The prime contractor was Software Productivity Solutions, Inc., Indialantic, FL, with subcontractors from General Research Corporation, Santa Barbara, CA and VeriQuest, LLC, Raleigh, NC.

With the downsizing and reorganization within the Government, the pressure is on to demonstrate transferable, usable technologies. To facilitate technology transfer of certification technology, RL initiated a Memorandum of Agreement (MOA) with the Gunter Annex of the Maxwell Air Force Base, AL, through Ms. Judy Roberts, Program Manager of the Air Force's Reuse Center (RC). Under a separately funded project, Gunter is planned as a beta test site for trial use of the certification technology developed under CRC. Also, as a separately funded project, RL has an agreement with UL to provide feedback on the innovations developed under CRC. Underwriters' Laboratories' (UL) and Gunter's planned participation will validate the underlying ideas while providing valuable information for enhancement, refinement and continued exploration.

4.2 Project Goals

The CRC project goal is to make certification usable, practical, cost-effective, and measurably beneficial. Primary project activities include development of a Certification Framework, implementation of a prototype automated certification environment, and demonstration of the framework and prototype environment through application by pilot users. The framework development was an incremental process, with refinements based on feedback from pilot users.

To meet the overall goal of usability, practicality, and cost-effectiveness, the program established specific objectives as follows:

1) Select only a practical, usable, and cost-effective subset of reliability and quality techniques that can be demonstrated to improve confidence in reusable software.

   To meet this objective, data was collected from an extensive tool survey and evaluation, studies of faults and testing and analysis techniques, and pilot user feedback.

2) Synthesize these techniques into a cohesive framework that is sensitive to different user requirements.

   To meet this objective, the framework was designed for adaptation. Adaptability was demonstrated by deriving processes for certifying the quality concern of correctness for code assets and then extending the framework to assess architecture assets.

3) Make certification understandable, practical, and usable for the typical engineer by hiding the theories and complexities.

   To meet this objective, the prototype Automated Certification Environment (ACE) was designed to guide the user through the selection process, viewing straightforward cost and benefit assessments and automatically selected lists of techniques and tools. The environment integrates usable certification tools that can be readily applied by software engineers of average skill levels.

4) Reduce the cost of reliability and quality improvement techniques.

   To meet this objective, the program focused on making costs known and relating these to quantified benefits. In addition, the prototype environment integrated certification tools in such a way that the certification process is made easier and more productive.

5) Design a cost-effective certification process in terms of quantified costs and benefits.

   To meet this objective, the program developed a cost and benefit model which presents the cumulative costs and benefits of applying automatically suggested techniques and tools.

6) Refine and demonstrate a piece of the framework that is demonstrably usable, pragmatic, and cost-effective for near-term application.

   To meet this objective, the program conducted an initial certification field trial and is working with pilot user sites to apply and refine the framework for the certification criteria of correctness, a key concern for reusers.

4.3 Measures of Success

Early in the project, several measures of success for CRC were defined as appropriate for a research and development project. Figure 4-1 illustrates how the measures of CRC success for R&D projects span three areas: Innovation, Experimentation and Validation.

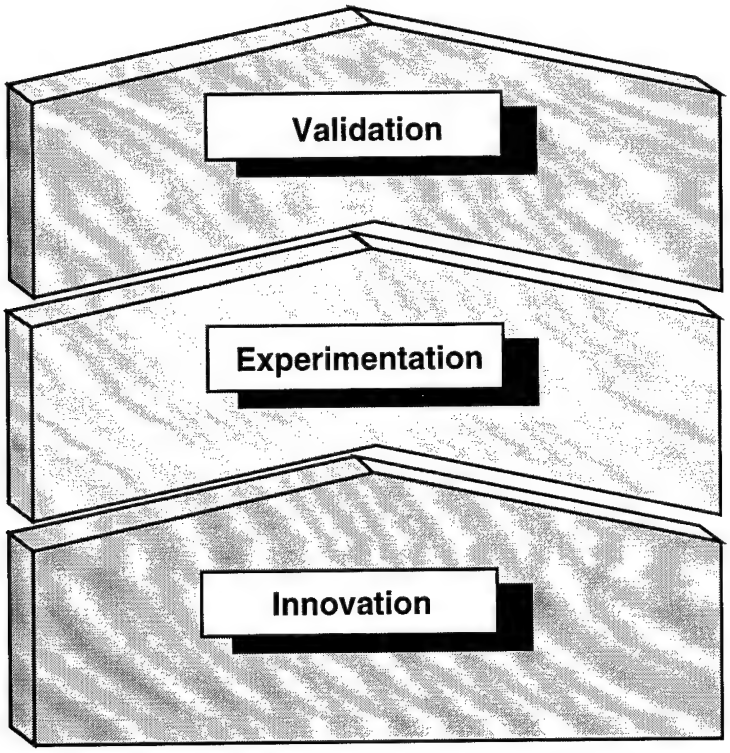


Figure 4-1. Measures of success as an R&D project

The innovation "wedge" of this upwardly progressing arrow consists of our innovations in theoretical developments. Experimentation, the second wedge, is the application of our theory to a laboratory environment. The third wedge, Validation, is achieved through application of the innovations to a real-world situation. Assessments against each of these three measures were performed at the end of the project and are reported in the conclusions of this document.

In addition, measures of CRC success for technology transfer (both public and commercial sectors) were also established. The measures of CRC success for technology transfer span three areas: Awareness, Communication and Application as illustrated in Figure 4-2.

Awareness includes requests for information from users; communication includes bi-directional information exchange between RL and users; and application includes pilot site participation to apply the CF. Assessments against each of these three measures were performed at the end of the project and are reported in the conclusions of this document.

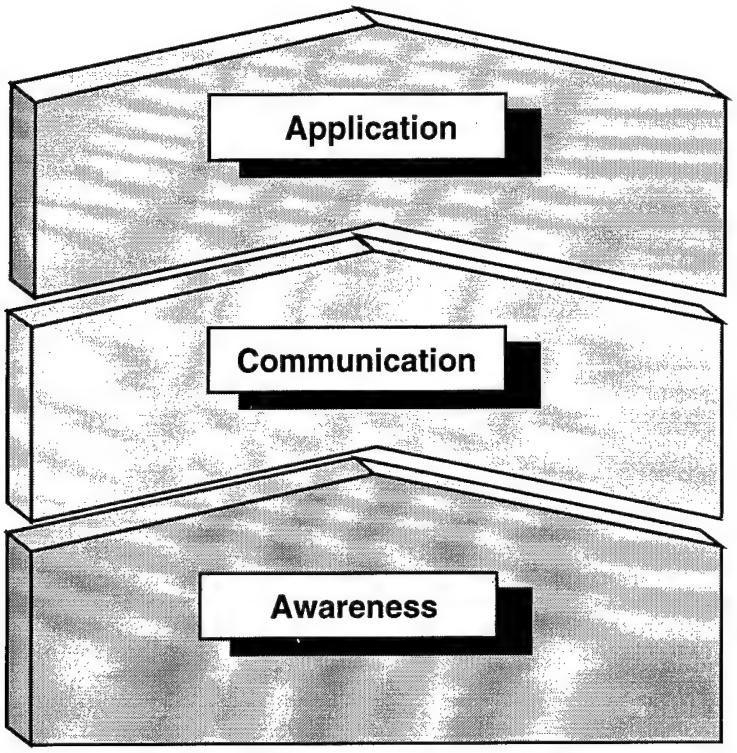


Figure 4-2. Measures of success for technology transfer

4.4 Technical Approach

Our technical approach to developing the CF had several driving forces:

- Concepts discussed in the CRC proposal
- Lessons learned from the Software Quality Framework (SQF)
- User needs and reuse scenarios
- Cost/benefit of certification

Each of these forces are discussed below.

Our proposed approach to achieve our project goals focused on "a desired level of confidence in a minimum level of required reliability." To achieve this, we chose to develop the CF as a tailorable roadmap for identifying certification requirements, objectives, tools and techniques. We selected techniques and tools based on usability, practicality, and effectiveness based upon empirical evidence. We integrated available tools to provide stable, high-quality, automated support.

Another driver in our technical approach was to develop the Certification Framework having learned from our experience with the Software Quality Framework (SQF). The

SQF has a rich, 20-year history from which to learn, and SPS was keenly aware of the technical obstacles and opportunities in the SQF and the CF. Both the CF and the SQF address a large, complex and multi-faceted problem with many associated issues. Both were pioneering projects at their start.

To address these issues, we reviewed a broad range of quality concerns including the SQF quality factors and identified "correctness," "completeness," and "understandability" as fundamental concerns. Correctness was defined as an absence of defects or non-conformance. For these quality factors, non-conformance was defined as latent defects, existence defects, and standards violations. Through our experiences with the Software Quality Framework (SQF), we developed a certification algorithm using a generic series of simple steps that could be tailored to users and their assets. We focused the CF on automated solutions that minimize manual activities and hide complexity to ensure its success. Selecting code as an asset type from the MIS (Management Information Systems) domain and correctness as our fundamental concern for our initial field trial, we applied the CF, collected and analyzed the results. A recommended next step for future funded projects is to identify other assets (e.g., architectures) and quality concerns (e.g., robustness) and to develop a plan for implementing the CF using these new attributes.

Working with potential users to understand their needs and constraints, we refined our technical approach. We developed an operational concept based on our certification algorithm to guide the user through the CF and automation tasks. We developed and refined a practical CF that can be tailored and extended to apply to all types of assets. We demonstrated automation with incremental, prototype certification demonstrations. We plan to monitor pilot use of the CF and collect feedback from these experiments to validate and refine our technical innovations.

The technical approach to developing the CF was also driven by a cost/benefit perspective. We developed a cost/benefit model to guide the users' selection of certification techniques. The cost includes incremental investment cost for establishing the certification process and the incremental cost of executing the certification process. The benefits are reduction of risk from reuse (i.e., rework avoidance) and an increased attractiveness of reusable assets. The CF employs activities that are both cost-effective and valuable to users of assets. The cost/benefit models, together with the CF, support decision-making.

4.5 Task Areas, Roles and Work Products

CRC's four major task areas, the individuals responsible, and the associated work products are shown in Figure 4-3. All of the project documents support this document, Volume 1, Project Summary. It is recommended that Volume 1 - Project Summary be read first, followed by the supporting documents in their numerical order. However, the reader may also skip to documents of particular interest, after the Project Summary is completed.

As shown in Figure 4-3, the Certification Framework Development (Versions 1.0, 2.0 and 3.0) is found in Volume 2 of the document suite. The Cost/Benefit Plan is Volume 3. The Operational Concept Document (OCD) is Volume 4. The Certification Field Trial (Procedures Guide and Results) is found in Volume 5. Tools evaluations and the code defect model, both supporting the development and execution of the field trial, are found in Volume 6 and Volume 7, respectively. The Automated Certification Environment Prototype is documented in CRC's OCD, Volume 4.

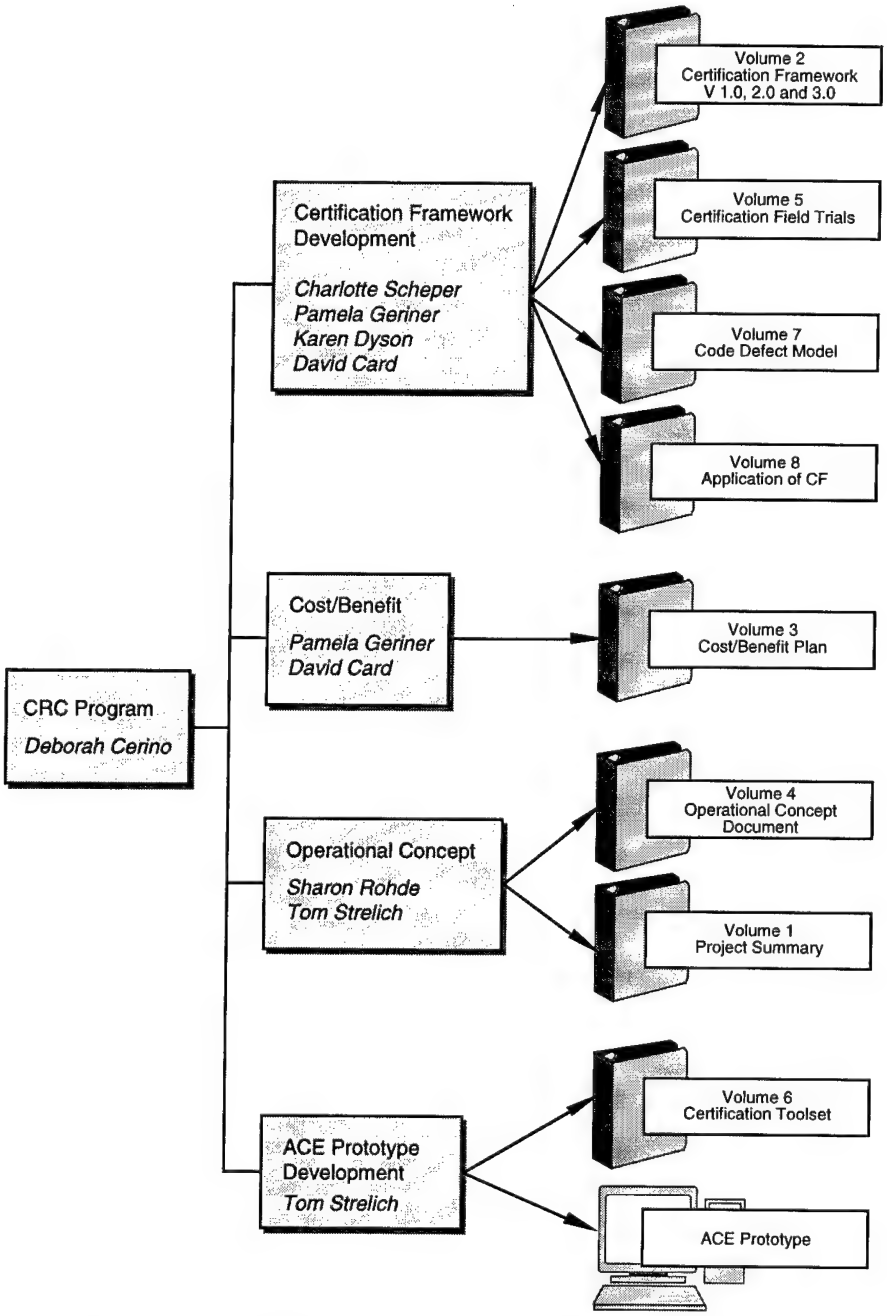


Figure 4-3. CRC's major task areas and products

5 Reuse and Certification Technologies

For both reuse and certification technologies, we assessed the state of the art (i.e., theory) through a survey of previously published research literature, and we assessed the state of the practice (i.e., application) through industry interviews. In addition to these resources for our assessments, the staff of the CRC team has many years of collective, professional experience in both of these areas.

Our findings indicate that both of these technologies are immature. The results of our assessments bounded the scope of the CRC project, refined our technical approach, and ultimately drove our development of the CF. This section begins with a discussion of the history of reuse (i.e., "where we've been"), continues with a collection of selected experts' opinions about the state of the art and state of the practice, and ends with an examination of the trends in reuse and certification.

5.1 History

Reuse is a central principle of science and engineering that dates back at least to the establishment of the first learned societies and scientific journals in the 17th century [POO92]. Societies and their journals of those days served to recognize members for their contributions and to function as quality control for the community. Publishing in journals established a record of work accomplished and provided an open forum for public review. The concepts from these reviewed works could be "reused" for the further advancement of knowledge. For example, when solving a problem, a known solution is applied to similar new problems. If only some elements of the solution apply, the solution is adapted to fit the new problem. Proven solutions, used over and over to solve the same type of problem, become accepted, generalized, and standardized. Towards the end of the 19th century, records of this type of knowledge had become so large that most developed disciplines of the time began establishing a collection of standard information that is now central to the endeavor of most scientific and engineering disciplines.

Nonetheless, the idea of software reuse is usually credited to McIlroy in 1968 when he presented his paper at the now historic NATO Conference [MCI69]. McIlroy, who originally coined the term "reuse," wanted to change the "craft" of software engineering into the "industry" of software engineering. It was not until the early 1980s that several major advances in the area of software reusability originated in research groups and industrial reuse projects in Japan (e.g., Hitachi, Toshiba), the U.S. (e.g., the STARS program) and Europe (e.g., ESPRIT). Several workshops and conferences focusing on this topic were initiated in 1983, and new thinking began to evolve.

During the early stages of reuse research, much effort was invested in reuse libraries (i.e., the classification of components, how to navigate in such repositories, how to compose new software out of predefined components, etc.). In retrospect, some researchers have pointed out that focusing on repository issues and component

collection skirts other important issues that are barriers to reuse. Consequently, researchers are investigating reuse from the perspectives of process-driven, domain-specific architectures, improved methods and tools, and economic return-on-investment rather than reuse libraries. Today, software reuse has now become an independent area within software engineering, emphasized by special tracks at most software engineering gatherings within industry and its own annual conferences and symposia.

5.2 State of the Art

Although there have been intensive research attempts and industrial projects in software reuse for about 15 years, the software industry still suffers from long time-to-market, low quality, and low productivity. Reuse is still far from being implemented as an integral part of software engineering. Reuse certification practices vary significantly, and the state of the art remains immature. Several researchers have recently assessed the state of the art of reuse and certification and a summary of their findings follows.

Card assessed the state of the art of component certification and discussed the following findings in a technology report funded by Rome Laboratory of the U.S. Air Force Materiel Command [SPS94]:

- Methodologies to implement reuse have not been fully developed.
- Tools to support a reuse process are lacking.
- Standards to guide critical software reuse activities have not been established.
- The process of assessing reusable components varies significantly and remains immature.
- The current state of reuse certification technology can only accommodate source code assets, even though domain-specific architectures may have the potential for high pay-off reuse.
- Automated software tools are used without validating their effectiveness.

One area that requires more research is certification criteria. Current reuse programs have indicated that their main concerns are centered around the criteria of completeness, correctness, understandability, and modularity.

Gall believes that reuse lacks standards, and future research should be focused on building these standards to mature the discipline of software engineering [GAL95]. Gall believes that a catalogue of standardized software components will facilitate incorporating reuse as part of the software development process. He used the analogy of other classical engineering disciplines such as electrical engineering, where "reuse" of standard parts is routine and concluded that concentrating on what parts to include

in a standard catalogue of a reuse library is more useful than looking at how to administer the components that the libraries house.

Gall also believes that software engineering lacks formality, and consequently, reuse suffers. Formal-based approaches and solutions are viewed as too mathematical and therefore, not accepted by industrial software engineers. However, Gall feels that the emergence of domain analysis may overcome many of the problems of software reuse. Domain analysis needs to be both formalized and combined with other software support areas to become relevant to the software development in industry.

Gall points out another hindrance to reuse is the limited interactions of the reuse community with other related communities (e.g., maintenance, object-oriented). This tendency is true with many specialized groups within software engineering. Instead of openness to other related research areas, the software reuse community tends to be a "closed" group. Each community has its own conferences and many researchers work as if their problems were unique. Gall believes that strengthening communication among related research communities will significantly improve research results in software reuse, and enhance the integration of reuse into a broad development methodology.

Prieto-Diaz assessed the current state of the art of reuse and identified areas for future research [PRI93]. Among those he identified are the need for certification, valid economic reuse models, and integrated tools and object-oriented methods to span all development phases.

Samadzadeh created a list of lessons learned from his experiences and serves as an indirect assessment of the state of the art [SAM95].

1. Reuse is a management decision.

   Even though the rewards of reuse are great (i.e., 70% reduction in time to delivery), the cost of institutionalizing reuse is substantial (i.e., 30-50% increase in development costs). Therefore, management must be involved in providing resources and direction to incorporate reuse into an organization's way of doing business. The major inhibitors to reuse are non-technical and can best be eliminated with management intervention.

2. The reuse rules of three are true.

   Reuse requires domain knowledge in order to recognize "what to make reusable" and "how to make it reusable." The break-even point for recovering the additional investment cost turns out to be around three reuses of a particular component. These are based on Ted Biggerstaff's proposed "Reuse Rules of Three" and developed by Bob Alanergans's observations at Raytheon.

   a) Before you can develop reusable software you need to have developed it three times.

b) Before you can reap the benefits of reuse, you need to reuse it at least three times.

3. You need to reuse more than just code.

   Code reuse is the easiest software artifact to reuse, but in order to achieve large improvements in software productivity, one must reuse other portions of the software development life cycle. Pushing reuse earlier and earlier into the software development life cycle has the highest payoff (i.e., in the requirements phase). Reuse of domain-specific software architectures has high payoff.

4. The "glue" is the key technology.

   Experience has shown the need for common communication protocols and simple, well-documented integration mechanisms to bring about reuse of software components. Putting components on the shelf is only part of the problem, one needs the right glue to put them together with. CORBA (Common Object Request Broker Architecture) provides an integration mechanism that shows promise for component reuse.

Samadzadeh feels that the reuse community can benefit from his lessons learned and should focus future research to address these aspects of reuse.

The overall conclusion of the studies seems to be that the state of the art is still somewhat immature . The concepts underlying reuse and certification technologies are still far from being implemented as integral part of software engineering practice.

The underlying premise for certification is that it should increase user confidence in the quality of reusable assets. Uncertainties about the quality of reusable software present real risks and, as a result, are serious impediments to increased reuse. All of the Government-sponsored reuse repositories are applying some certification process in an effort to reduce risks and increase reuse. However, very little is known about the effect of these certification processes on reuse or the ability of the processes to assess quality. Moreover, there are increasing concerns about the cost of certification, which ranges from one to several person-weeks per asset. Further research is needed in certification processes, techniques, and tools in order to identify the cost-effective approaches.

5.3 State of the Practice

The Defense Information Systems Agency (DISA) surveyed a group of repository personnel and experts on reuse [DIS94]. They found that there was very little empirical data on either code asset or non-code asset reuse. In addition, the findings showed that 70% of the respondents agreed that reuse certification is a necessary activity, 45% were not aware of any standards or do not use standards in their certification, and 90% did not know the actual cost of certification. The respondents felt reuse certification should provide:

- a review of a reusable asset to determine its suitability for reuse, what is known about the asset, and what accompanies the asset
- a check for completeness and reliability of the asset
- a form, fit, and function check and some kind of confidence level that the asset works in its appropriate context
- a check to ensure the asset complies with standards
- the assurance that the asset has been checked against a series of analysis techniques and the outcomes have been documented
- the definition of the use and context of the asset (domain) to give correctness to the asset
- information that will help users decide the closeness of fit

This DISA study recommended several actions; the first being that costs and benefits of certification be evaluated in order to provide DoD advice on future resource allocations for certification activities.

Prieto-Diaz assessed the state of the practice of reuse and believes that the problem we face in software engineering is not a lack of reuse, but a lack of widespread, systematic reuse [PRI93]. He feels that reuse of code, subroutines, and algorithms, as well as reverse-engineering, is widely practiced; but it is done informally, and on an ad hoc basis. This informal practice, in which components are selected from general libraries, is usually called opportunistic reuse and is very much the state of the practice today. Reuse is conducted at the individual, not the project level; procedures for reuse don't exist; and the libraries in use contain components not designed for reuse. Nonetheless, Prieto-Diaz believes that the near future will see significant progress, and he is hopeful that reuse will become institutionalized. Reuse, in the end, should come "so naturally that we do not have to think about it [PRI93]."

Several other studies have investigated the issue of certification for reuse in order to assess the state of the art and the state of the practice. Development reuse organizations reported that they wanted the Government "to 'certify' the testing level that a component has undergone and the reliability of the component so that a contractor does not have to duplicate similar testing procedures" [BUN94].

Frakes assessed the state of the practice by surveying 113 people from 29 U.S. and European organizations asking 16 questions about reuse [FRA95]. Frakes admits that the survey respondents do not form a large random sample of the software engineering community. However, he believes that indicators of the experience, education, and background of the respondents suggest that his sample is fairly representative of

experienced software engineers and managers at high-technology companies. The results of his survey are shown in Table 5-1.

Table 5-1. Answers to sixteen reuse questions [FRA95]

	Questions	*Answers*
1.	How widely reused are common assets?	Varies
2.	Does programming language affect reuse?	No
3.	Do CASE tools promote reuse?	No
4.	Do developers prefer to build from scratch or to reuse?	Reuse
5.	Does perceived economic feasibility influence reuse?	Yes
6.	Does reuse education influence reuse?	Yes
7.	Does software engineering experience influence reuse?	No
8.	Do recognition rewards increase reuse?	No
9.	Does a common software process promote reuse?	Probably
10.	Do legal problems inhibit reuse?	No
11.	Does having a reuse repository improve code reuse?	No
12.	Is reuse more common in certain industries?	Yes
13.	Are company, division, or project sizes predictive or organizational reuse?	No
14.	Are quality concerns inhibiting reuse?	No
15.	Are organizations measuring reuse, quality, and productivity?	Mostly no
16.	Does reuse measurement influence reuse?	No

Frakes found that while some common reusable assets, such as the UNIX tools, are widely used and highly regarded by software engineers, others are not. It was refreshing to see that most software engineers would prefer to reuse software rather than build it from scratch. This result contradicts the commonly-held belief that software engineers prefer developing new code themselves rather than reusing.

Frakes also found that programming languages and CASE tools do not seem to have an affect on reuse. However, reuse education is important for improving reuse. Survey results indicate that a common software process promotes reuse.

On the other hand, Frakes' findings indicate that software reuse does not increase with software engineering experience, nor do legal problems cause a serious reuse impediment. Reuse levels were significantly higher for life cycle objects in some domains (such as telecommunications), rather than others (such as aerospace). There was no relationship between organizational size and levels of reuse. His respondents were not influenced by concerns about asset quality. Few organizations measure reuse, quality, and productivity, even though measurement is needed to manage a systematic reuse program.

The immaturity of the state of the practice as shown in these industry assessments was confirmed by our project surveys through on-site interviews of Government repository community. Our assessment of the state of the practice of certification for reuse is based upon interviews with 22 individuals at 6 different sites. We found that certification is resource-constrained. One to two weeks per asset seems to be the maximum staff-effort allocated to a certification activity in industry practice. The staff members employed by repositories in industry have little domain expertise and weak testing expertise. Users and repository staff like a "level-orientation" for certification activities. Additional details of the user interviews can be found in CRC's OCD.

In light of all these recent assessments, why isn't reuse where it should be? Researchers are quick to point the finger at one area and say that reuse is not happening because of "X." Examples of "X" are lack of components, symptoms of Not Invented Here (NIH), poor quality, no business strategy that encourages reuse, and lack of domain architectures. More than likely, reuse is not happening due to a combination of many factors, and each operating with differing weights for each particular situation.

Polls have been taken to ascertain which areas are the "culprits" and the rationale is to attack those. These findings are only as good as the population sampled, and it is nearly impossible to sample all those who are involved in reuse in some way across all domains (it may not be called reuse). Reuse may be more appropriately called "working smart" or "good engineering" building on expertise.

5.4 Trends

Today, software reuse is no longer in its infancy, yet it appears that little progress has been made. However, a report developed by Boeing on the STARS program indicates that this situation appears to be steadily changing [BOE93b]. The barriers that have inhibited reuse are gradually diminishing and, due to a downturning and more competitive economic climate, it is becoming increasingly critical for organizations to overcome those barriers. The result has been a recent substantial increase in the number of industry and government initiatives focusing on establishing reuse projects. Several efforts are now underway to promote reuse of software across these agencies and within industry.

Hooper feels that reuse concepts are moving from research into practice, and very good results are being reported [HOO91]. An initial investment in reuse (i.e., organizational changes, initial library development, training, etc.) is required. There has been an understandable reluctance to make this investment without reasonable assurance of success. Hooper believes that enough reuse successes are accumulating to allay the concerns; thus, he expects an increase in the number of organizations undertaking the practice of software reuse.

Increased reuse activity has also built upon advances in the theoretical and practical foundations for reuse. Among these are domain analysis and domain engineering as distinct fields of study, reuse library technology, process improvement, and Total Quality Management (TQM) principles. Some of these are still immature, but their growth has helped advance the maturity of reuse and certification technologies.

6 Reuse Context

Given the state of the art, state of the practice and future trends of reuse and certification technologies, the CRC team established the conceptual context of our project work. We defined our technical bounds by group consensus and constructed a context diagram which describes the realm in which we operate. This realm of operation, or the reuse context, is the set of circumstances and requirements within which reuse is carried out.

Since certification is just one part of the overall reuse process, it is necessary to determine which elements of that context affect certification and which elements are themselves affected by certification. It is also necessary to determine what role the context elements play in certification so that the certification process can be designed to be sensitive to that context and adapt as necessary to its requirements and circumstances. Figure 6-1 depicts a conceptual diagram of the reuse context in terms of the elements and processes by which it is defined, i.e., our reuse context for asset quality certification.

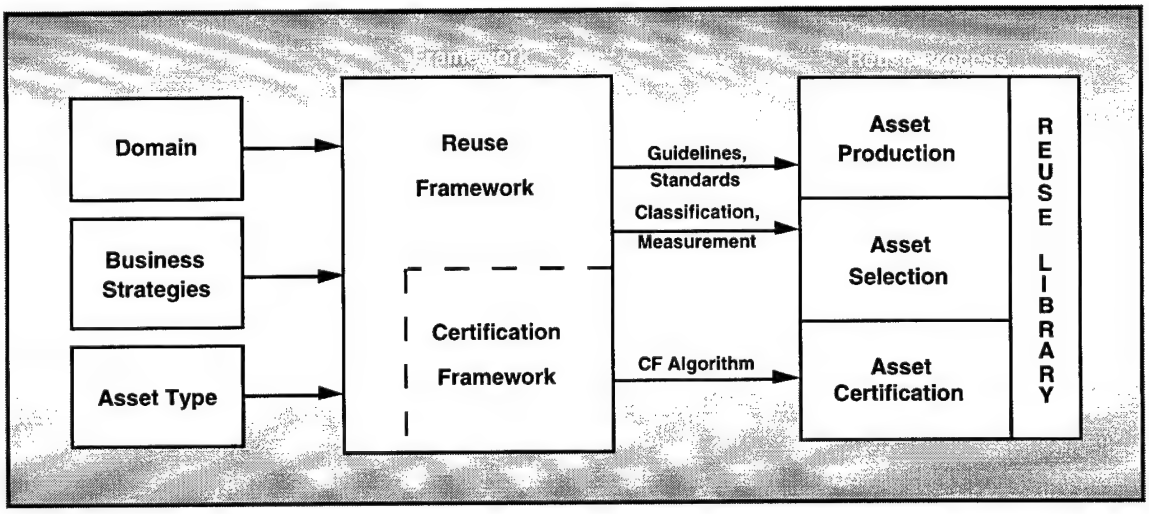


Figure 6-1. Reuse context for asset quality certification

As seen in Figure 6-1, the elements of Domain, Business Strategies, and Asset Type drive both the Frameworks for reuse and certification. The term "domain," as used in our reuse context, can refer to either the application domain for which an asset is developed or the application domain in which the asset will be reused. To determine the particular domain for a reuse context, a domain analysis should be performed.

As illustrated in Figure 6-1, the Reuse Process is composed of three subprocesses: Asset Production, Asset Selection, and Asset Certification. Asset Production is the process by which an asset is developed and made ready for inclusion in a library or repository. Asset Selection is the process by which a potential reuser searches the library and selects candidate assets for use in a new system. Asset Certification is the process by which an asset is evaluated for conformance to the requirements it must satisfy to be reused. Asset Production, Asset Selection and Asset Certification are basic reuse activities within the Reuse Process and all may be performed using an associated Reuse Library.

The specifics of what these processes need to accomplish are largely determined by environmental elements of Business Strategy of the library and the Domain(s) for which the library's assets are intended. The details of the processes and the procedures for performing them are defined by the Reuse Framework through guidelines, standards, classification schema, measurement, and other necessary mechanisms. The Certification Framework is a subset of the overall Reuse Framework and defines the certification process.

Within the reuse context, the focus of CRC was upon a subset of the Reuse Framework called the Certification Framework (CF) as well as its algorithm for asset certification. The CF is influenced by some of the same inputs as the Reuse Framework. As an output, the CF Algorithm is used to certify assets subject to the CF (and the Reuse Framework). The context diagram helped the CRC team to identify the scope of the problem and focus our project on a portion of the problem that was doable within CRC's time and effort. Prior to CRC, very little prior work had been done to establish a certification framework, yet its development is influenced by the other elements of the reuse context. Research studies, both theoretical and empirical, related to asset quality certification are documented in Volume 2, the Certification Framework and Volume 4, Operational Concept Document.

Yet, we also observed that many of the elements of decomposition within the reuse context have undergone many years of previous research. Because these elements established an underlying, technical foundation for our reuse context, a survey of literature was conducted to provide background information for five areas: domain, business strategies, asset production, asset selection (asset classification and schemas) and reuse frameworks. Through our literature survey, we critically evaluated the past research in order to focus our work and "do what made the best sense." This survey provided the technical foundation for our project and its results impacted our development of the CF. Consequently, we were able to established a clear research direction for CRC and ensure that we did not duplicate work from prior projects. The results of this task are an annotated bibliography for each of these five areas which appears in Appendices A-E of this document.

Table 6-1 tallies the number of articles reviewed in each of these topic areas.

Table 6-1. Total research articles in each of the background areas

Topic	Number of references
Business Strategies	16
Domain Analysis	27
Asset Production	38
Asset Selection	12
Reuse Frameworks	17
Asset Certification	9

The following subsections discuss the findings in the first five topic areas.

6.1 Business Strategies

As recorded in Table 6-1, a few articles have been written about business strategies for reuse, but not nearly as many as the other researched areas. Authors have begun to define business problems associated with reuse and certification and have identified associated issues. For example, Banker believes managing reuse requires monitoring the firm's software at the organizational or enterprise level rather than at the traditional individual software project level [BAN93]. Card suggests that reuse fails because organizations treat reuse as a technology-acquisition program rather than a technology-transition problem; organizations fail to approach reuse with a business strategy [CAR94]. Jones indicates that the most sizable payoffs across types of reuse occur after 36-48 months; organizations must plan for payoffs in the out years [JON95]. Currently, the aspect of business strategies for reuse is seen as a necessary ingredient in the reuse context, largely because previous reuse projects have been disappointing or have failed.

Of significant interest to CRC in the area of business strategies was the definition of business strategy archetypes to support a Software Reuse Business Model [DIS95]. A software reuse business model (SRBM) was developed by the U.S. Army Space & Strategic Defense Command to provide a structure to define typical architectures and implementation plans of organizations within the DoD that are in the software reuse business. The SRBM supports engineering activities, business planning, and contracting activities. It details both the perspectives of "practitioners" who are responsible for reuse and the activity flows in software system acquisition, including the inputs, outputs, controls, and mechanisms for each activity. The SRBM provides an acquisition view of libraries of reusable assets rather than an economic view from the contracting organization or from their developers. The SRBM is designed for reuse and does not discuss certification. The U.S. Army Space & Strategic Defense Command also developed detailed procedures to evaluate components for reuse and can be found in a supporting document to the SRBM [ARM95].

As part of the model, a set of eight generic business strategies, called "archetypes", were developed:

- Vendor-owned domain
- Government-supported standard
- Value-added reseller
- Government-owned architecture
- Government-owned domain
- Reengineering
- Public library
- Commercial library

These archetypes were defined from the acquisition perspective and vary in the degree of control that Government and industry have over the definition and implementation of reusable assets. For example, in the *vendor-owned domain*, control is completely in the hands of industry; in the *government-supported standard* and the *value-added reseller*, there are increasing levels of government influence; in the *government-owned architecture*, control is shared by government and industry; in the *government-owned domain* and the *reengineering*, control is completely in the hands of government. The *public library* and *commercial library* archetypes provide a repository for assets over which they have control and for which they have to recover costs, the public library through independent funding and the commercial library through fees. Additional details on the SRBM and other published literature in the area of business strategies can be found in Appendix A.

6.2 Domain

As seen in Table 6-1, much has been written about domain analysis, but experts do not firmly agree upon how to do domain analysis. As shown in Figure 6-2, research in domain analysis has been on-going for over 15 years and authors and their methods have different historical roots. Neighbors investigated domain research topics which ranged from theoretical to empirical; these research results sparked the other studies shown in Figure 6-2.

Experts in the field over this 15-year period include Arango, Bailin, Batory, Lubars, Neighbors, Parnas and Prieto-Diaz. Research projects and company projects include CAMP, DRACO, GENESIS, IDeA, RLF, NASA, SEI, STARS, GTE, Unisys, CTA, UC-Irvine and UT-Austin. Hess and his colleagues compiled a significant bibliography of domain analysis, as of 1990, as part of the Domain Analysis Project at the SEI [HES90]. Appendix B documents these authors and others who have been significant contributors to this body of work.

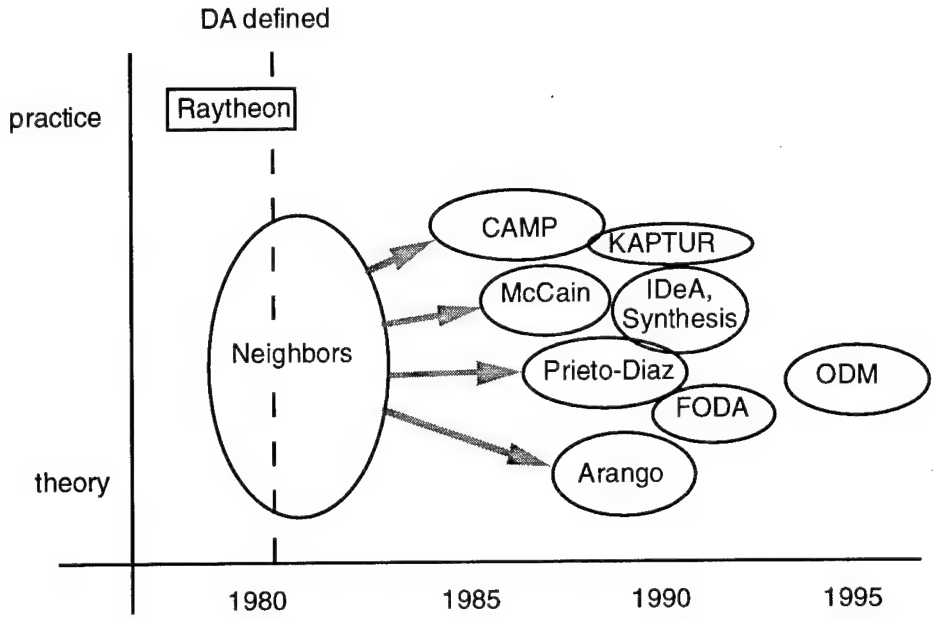


Figure 6-2. Time line of research in domain analysis

Four major domain analysis methods exist: the Prieto-Diaz method, Synthesis, Feature-Oriented Domain Analysis (FODA) and Organization Domain Modeling (ODM). The first method was developed by Prieto-Diaz and consists of identifying objects and operations. The Prieto-Diaz method uses abstraction, classification schemes and taxonomies similar to those of library science and zoology. Synthesis was developed by the Software Productivity Consortium (SPC) and emphasizes flexible production lines to drive the development of software. Synthesis leverages commonality among problems and adapts previous solutions to accommodate differences in new problems. FODA was developed by the Software Engineering Institute (SEI) and focuses on user needs and requirements to analyze a domain. And lastly, ODM was developed by Unisys and Organon Motives under the STARS program. ODM uses the processes of the Conceptual Framework for Reuse Processes (CFRP) funded by the Software Reuse Initiative (SRI).

Even though most researchers would not agree on which method to use, they *would* agree that domain analysis should be performed prior to developing a reuse program and reusable assets. In past reuse projects, this critical activity may have been missing and may have contributed to failures in reuse. The four methods of domain analysis have emerged in research and are slowly being applied in case studies for validation, refinement and documentation of lessons learned. Additional details on the topic of domain analysis can be found in Appendix B.

Identifying a domain in a reuse context is one of the important activities of domain analysis. Domain analysis is a needed activity to plan for reuse within an organization. Domain analysis will move reuse beyond ad hoc components salvaged from a collected library. Hess is quoted as saying "Domain analysis is the foundation for establishing a reuse program within an organization" [HES90].

In an organization's business plan, the software development process must accommodate both the development of software and the experience factory at the component level. Both have different objectives and perspectives. A business plan can augment informal sharing of code and associated experience between developers on a project. Studies reveal that some domain types are more appropriate for reuse. For example, computational modules are good candidates for reuse since they perform standard operations and are easily "plugged and played."

Using domain analysis, software application systems have been analyzed and categorized into many different sets of domains, but no agreed-upon standard taxonomy exists. For example, the DSRS categorizes software domains as Finance, Health, Human Resources, Reserve Components, Materiel Resources, Procurement, Information Management and Command and Control. Of particular interest to CRC were the categories for application domains established by the National Software Data and Information Repository (NSDIR): MIS, Avionics, Command and Control, Automated Test Equipment, Weapon Systems, Communication, Intelligence, and Process Control [NAT95]. More than likely, rather than creating an exhaustive list of all software domains, these categorizations were an outgrowth of the data currently housed at these agencies.

6.3 Asset Production

As shown by the count of research articles in Table 6-1, much has been published in the area of Asset Production. This may be due to reuse's initial focus upon source code and asset production, in an ad hoc or "grass roots" manner. Disappointing results of these early efforts led to a higher level, or organizational, view of reuse incorporating the techniques of domain analysis and a structured, tailored reuse business plan. Effective reuse is now seen as more than producing modules and storing them, in hopes that another may find them useful.

To produce reusable assets, the software must be designed with that characteristic as a requirement. Different levels of reuse are possible, ranging from source code to architectures. Standards groups can provide guidance for production of assets for reuse. Other artifacts need to be developed (e.g., reuser's guides) to support successful reuse. Reusers need to be supported in the decision-making process for "make" versus "buy" (i.e., should the organization invest in producing the asset, or is it more cost-effective to buy the asset from a commercial-off-the-shelf libraries of existing assets?). Asset evaluation and selection may be a multi-pass process as shown in Figure 6-3.

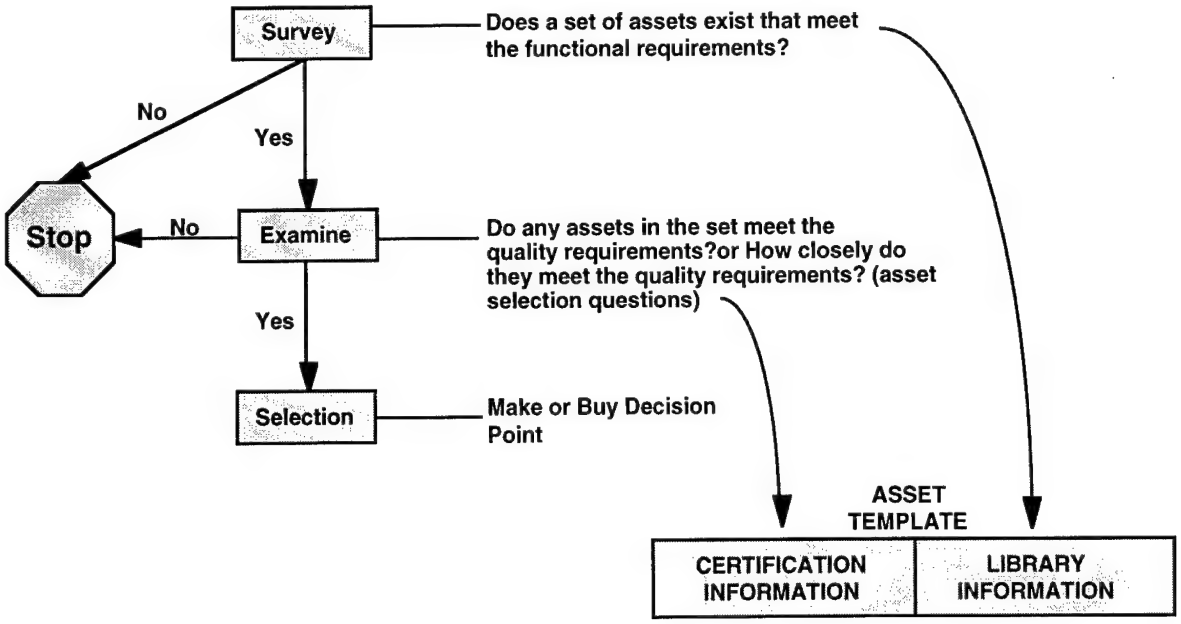


Figure 6-3. Asset evaluation/selection and the make-versus-buy decision

We must not overlook the simple fact that a component is not reusable unless it is used; this means that a reusable component should be built out of real needs. Needs include not only those of the current time, but also those potential future needs. Therefore, producing reusable assets should be targeted at products that fit the traditional economic model of supply and demand, that is, the producer-consumer model. An asset needs to be produced that fits the needs of those using it. Planning for the results of asset production cannot be made in a vacuum.

In an older reference of 1989, Biggerstaff reported that the productivity of the software creation process had increased only 3-8% per year for the last 30 years [BIG89]. Experienced software engineers know well the feeling of *deja vu* that is so characteristic of their trade. Several attempts have been made to measure this phenomenon, however, experimental conditions are difficult to control and quantify. Jones estimates that less than 15% of new code serves an original purpose [JON95].

From our literature survey of Asset Production, it appears that the technologies to support reusable asset production are maturing, and in the near future, will be better able to increase the productivity of developers in the software creation process. For example, higher-level languages, object-oriented design, and the techniques of encapsulation and levels of abstraction can enable software reuse. Reuse tends to increase with the use of program families consisting of building blocks with limited dependencies. Code templates and automatic code generators based on these design techniques shows promise for increased reuse. Likewise, a sensitivity to standards and quality factors makes components more attractive to reusers. Also, the technique of domain analysis to produce generic, process-driven architectures, together with

business planning, can lead to very large scale reuse. Additional details on the topic of asset production can be found in Appendix C.

6.4 Asset Selection

Not as much research has been done in the area of Asset Selection, since this is primarily a library science topic, rather than a traditional software engineering topic. How information is organized and catalogued is part of the science of information retrieval; experts agree that software staff could learn from the principles already established outside of their field. Inter-disciplinary communications and collaboration between software and library science could benefit and accelerate the maturity of reuse.

The World Wide Web (WWW) may be a promising mechanism for providing users a convenient access for selection of software components from multiple reuse libraries. Growth in the popularity of the Internet and the WWW, as well as the wide availability of WWW client and server software, has accelerated the shift from centrally maintained software repositories to virtual, distributed repositories. Now a "virtual repository" that catalogues software maintained by other repositories is possible. The main advantage of distributing a repository is to allow the software to be maintained by those in the best position to keep it up-to-date. Well-maintained software repositories are central to software reuse because they can make high-quality software widely available and easily accessible. Also, copies of popular software packages may be mirrored by a number of sites to increase availability (e.g., if one site is unreachable, the software may be retrieved from a different site and to prevent bottlenecks).

One such repository is Netlib, a collection of high-quality publicly available mathematical software. Netlib, in operation since 1985, currently processes over 300,000 requests a day. Netlib is serving as a prototype for development of the National HPCC Software Exchange (NHSE), which has the goal of encompassing all High Performance Computing Consortium (HPCC) software repositories and of promoting reuse of software components. Netlib was developed by Grand Challenge and other scientific computing researchers. Additional details about these types of commercial libraries can be found in CRC's Operational Concept Document.

Even though users can transparently access large volumes of components in multiple libraries, a critical aspect of the asset selection is related to the producer-consumer problem of economics. To be successful, software that is produced and put into libraries is must be useful to consumers; if the user doesn't need it, he won't select it. The user may ask these questions when selecting an asset:

- Does the asset meet my requirements?
  - Does it meet the functionality that it was built to meet (i.e., information supplied by the library)?

  - Does it meet the functionality for my intended application (requires a reuser's judgment)?

- To what degree does the asset meet my quality requirements (supplied by certification information)?
- What tradeoff can I make among these "degrees" for the set of quality factors (requires a reuser's judgment based on constraints)?
- Has this asset ever been used in a system like I am building?
- Who developed the original component (e.g., author, organization and standards used to develop the asset)?
- What is the cost and are there any legal rights restrictions?
- What standards were used by the library in certifying the asset (e.g., coding and style)?
- Are there any hardware or environmental dependencies?
- Are all the artifacts available?

Assistance in securing this kind of information would be valuable to the potential reuser. Additional details on the topic of asset selection can be found in Appendix D.

6.5 Reuse Frameworks

A modest amount of research information has been published in the area of Reuse Frameworks. A Reuse Framework may define the goals, plans and implementation by which an organization can accomplish reuse. For example, a Reuse Framework may guide the software developer in embedding reuse into their software engineering processes. Specifically, a Reuse Framework may provide procedures for capturing component design information and supporting rationale so that components can be assessed for future reuses. A Reuse Framework may also provide a reuse librarian with ways to administer a reuse library to support a project, an engineering group, an organization, or an agency. In addition, a Reuse Framework may provide a context for why reuse is or is not working.

However, among researchers and industry practitioners, no clear consensus exists as to what a Reuse Framework is and its role in the reuse process. The problem with establishing consistency and commonality with regard to Reuse Frameworks is complicated by the fact that the term framework can be applied at varying levels (i.e., organization, managers, developers, librarian, process groups, etc.).

DISA/JIEO/CIM (Defense Information System Agency, Joint Interoperability Engineering Organization, Center for Information Management), the DoD Software Reuse Initiative (SRI) and STARS (Software Technology For Adaptable, Reliable Systems) have provided the bulk of the research on reuse processes which can be attributed to the generalized category of Reuse Frameworks. This body of work and its concepts are mainly theoretical. On-going work needs to be conducted to apply these concepts and report the results as case studies. As these results are reported and the theoretical concepts refined, standards for Reuse Frameworks may begin to emerge, maturing reuse technology. The Reuse Interoperability Group (RIG) has been instrumental in establishing standards for asset certification techniques for reuse libraries. Additional details on the topic of reuse frameworks can be found in Appendix E.

6.6 Relationships Among Context Elements

From our research, we observed that relationships and dependencies exist among the elements of the reuse context as illustrated in Figure 6-4. For example, the domain is related to the business strategy and asset production. Business strategies are related to the reuse framework, asset production and asset selection. Asset production and asset selection are related by underlying principles. These relationships are explored in more detail in the following paragraphs.

These observations were confirmed by research findings. For example, a report of the Software Reuse Initiative (SRI) recommends that the development of the conceptual framework for reuse processes needs to consider the business aspect as well as the technical challenges to software reuse [DOD94a]. Tracz, in his article about reusability "coming of age" feels that organizations need to develop reuse programs that cut across domain, business type, and asset production [TRA87].

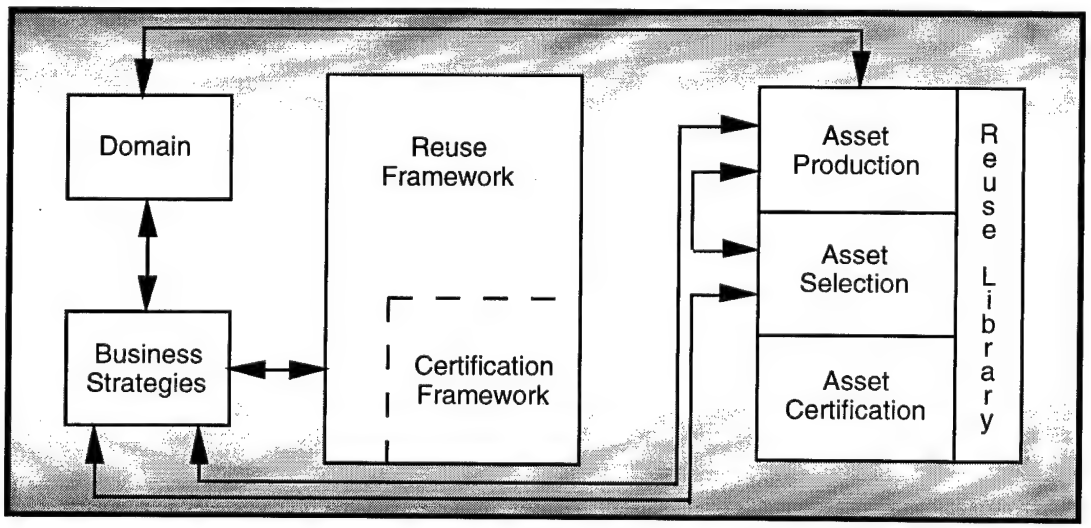


Figure 6-4. Relationships among elements of the reuse context

Likewise, the SRBM establishes a tie between the Domain and the Business Strategy of a library in their model [DIS95]. The authors state that "There is no single reuse business strategy that is appropriate to every system acquisition. A Domain Manager must formulate a strategy appropriate for systems in a given domain." The domain type influences the business strategy from an acquisition point of view, and a gray area exists between domain analysis and business strategies. Archetypes, which are derived from domain analysis, assist the domain manager in formulating business strategies. His resultant product lines congeal domain analysis and business strategies. Other underlying dependencies between the business strategy and other reuse context elements are identified in Table 6-2.

The Domain and Asset Production are related as follows:

- The activity of asset production is defined and planned for within a particular domain.
- Some domains lend themselves to reuse.
- Design for reuse within a domain architecture; layered abstractions encourage reuse.
- Asset designs should be useful across domains.
- Inadequate domain knowledge results in overly constrained designs.

Asset Production and Asset Selection are related as follows:

- The maturity of reuse process determines the kinds of assets produced and selected.
- Reusable building blocks are useless unless the designer knows that they are available for selection.
- The reuser needs to be able to find existing reusable software.

Asset Selection and Reuse Frameworks are related as follows:

- Even though a large amount of software exists in the files of software developers, most lack a large catalog to access those usable, reusable objects.
- Reuse efficiency and cost effectiveness requires a reuse framework.

Additional details about each of the five areas of research are found in the annotated bibliographies in Appendices A-E.

Table 6-2. Dependencies of business strategy on other reuse context elements

Element	Domain	Asset Production	Asset Selection	Reuse Framework
Business Strategy	• The domain is used in selecting a reuse business model. • The domain forms basis of cost and economic models. • Common functions of a domain are reusable in a business' product line.	• The activity of asset production is defined and planned for in a business strategy. • Asset production is best managed at the project and organizational levels for optimal reuse, assets are viewed as a capital investment. • During asset production, tradeoffs are made between the cost-effectiveness of make vs. buy. • High level reuse (i.e., design of architecture) has highest payoff.	• The process of asset selection is defined and planned for in a business strategy. • Successful reuse relies on effective search mechanisms to select assets within a library of useful assets. • When looking at a "used program," the business entity "selling" it is associated with the product selected. • Public domain assets are not free; the process of asset selection, cost and risk of searching the library for suitable assets, modifying the assets to fit the needs, and verifying their applicability may not be trivial.	• The needs of a reuse framework are assessed and planned for in the business strategy. • The reuse framework is part of reuse business products and process.

6.7 Validation of the Reuse Context

At the completion of the literature survey, we observed that this body of research validated our selection of elements in the context diagram of Figure 6-1. The CRC team selected the key elements for asset quality certification, and the reuse context diagram is substantiated by research.

We also discovered from our research that our view of the reuse context is unique, and therefore, innovative. To date, no other researchers have combined these significant elements in this way to construct a meaningful reuse context. Identifying the special aspects of each of these elements and how they influence each other, as part of a whole, is unrivaled and provides valuable insight into reuse and certification technologies. Our perspective paints an understandable picture of how certification "fits" within reuse and its associated elements.

7 Impacts of Reuse Context to the Certification Framework

Our established reuse context and the CRC research activities were a conceptual springboard for the development of the CF. Knowing the influences and dependencies of the elements within our defined reuse context, we developed the CF with a three-fold purpose:

1. Define the elements of the reuse context that are important to certification
2. Define the underlying models and methods of certification
3. Define a robust, decision support technique to construct a context-sensitive process for selecting the techniques and tools and applying them in order to certify assets

Table 7-1 summarizes the CF by listing the elements that compose it: the software reuse business model, domain, asset type, quality factor, non-conformance class, certification techniques, and certification process. For each of these elements, the table lists the possible attributes it may have in any particular reuse instance. The set of attributes for the software reuse business model, domain, asset type, and quality factor elements define the reuse context in which certification is performed. The non-conformance class and the certification techniques are selected based on this context and are used to tailor the certification process elements to the needs and requirements of this context.

Within this tabular view, the first four elements, the business model, the domain, the asset type and the quality factor determine a specific reuse context. A particular context is defined by the set of attributes chosen for each of these elements. For example, a specific reuse context could be defined by a software reuse business model of "Government-Owned Domain," a domain of "C^2"(Command and Control), an asset type of "Code," and a quality concern of "Correctness." The rationale for populating the first four elements of the CF with specific attributes is described in the following sections. Specific examples are included to clarify how the CF can be used for selected element attributes.

7.1 Software Reuse Business Model

For the element of the Software Reuse Business Model in Table 7-1, the eight archetypes developed by the U.S. Army Space and Strategic Defense Command were adopted to populate this element [DIS95]. These archetypes represent variations in the degree of control that Government and/or industry have over the definition and implementation of reusable assets, and thus over the decisions affecting certification requirements and justifiable cost/benefit ratios. The impact that the software reuse business model has on the certification process is significant, resulting in variations due to changes in responsibility and demands for quality, ownership of the artifacts used to assess the quality of the assets, existence of experienced staff, and the ability to pay for a level of certification commensurate with the certification needs.

Table 7-1. Tabular view of the CF

S/W Reuse Business Model	Domain	Asset Type	Quality Factor	Non-Conformance Class	Certification Techniques	Certification Process
Vendor Owned Domain	MIS	Design Information	Correctness	Latent	Compilation	Process Definition
Gov't Supported Standard	Avionics	Document	Completeness	Robustness	Static Analysis	Procedures
Value-Added Reseller	C^2	Test Artifacts	Understandability		Inspection	Tools
Gov't Owned Architecture	Automated Test Equip.	Req. Specs	Performance		Testing	Data Collection
Gov't Owned Domain	Weapon Systems	Code	Fault Tolerance		Formal Verification	Certification Levels
Reengineering	Communication	Architecture	Functionality		Benchmarking	
Public Library	Intelligence	Database Schema	Maintainability		Modeling	
Commercial Library	Process Controls	Models	Portability		•	
	•	Video	Reliability		•	
	•	•	Usability		•	
	•	•	Safety		Other	
	Other	•	Security			
		Other	Availability			
			Testability			
			Survivability			

7.2 Domain

For the element of Domain, the categorization of domains listed in Table 7-1 were selected for the CF because they are typical DoD domains and were adopted by the U.S. Air Force's National Software Data and Information Repository [NAT95]. Different domains have different certification requirements. Certain quality factors, such as performance, correctness, reliability, fault tolerance, etc., are more important in some domains than in others. Also, domains can have different expectation levels of importance for each of the quality factors. Domains where life-critical applications are common require rigorous determination of correctness, reliability, and safety. In such domains, system failure rates of no greater than 10^{-9} are usually expected. Domains where less critical applications are the norm might expect failure rates of 10^{-3}, whereas some domains might not have any failure requirement at all. In the first case, it is essential that there be no errors in the system components, whereas in the latter cases, some level of error could be tolerated.

When viewed as an aggregate, these differences in domains essentially specify different levels of certification, because they require different levels of rigor and thoroughness in the evaluation processes used in certification. The acceptable level of certification is generally determined by the intended application or use of the component. Generally, the more critical the component's correctness to system operation and the more harmful the effect of system loss or misfunction, the higher the required certification level.

The types of components and corresponding artifacts (supplementary information, documents, other assets, etc.) can also vary according to domain. For example, in domains characterized by embedded or parallel applications or by architectures with special features necessitated by dependability, requirements such as fault tolerance, documentation and/or models of characteristic hardware components might be included. Likewise, libraries for domains that are heavily database-dependent might include characteristic databases.

As an example of domain influences, consider a scenario where hardware and software for a complex space application is being designed. This application requires large amounts of numerical processing, large data bases, and iterative approximations to optimal solutions. It has demanding reliability and throughput requirements which require fault tolerant distributed or parallel hardware architectures. The mission that this application addresses requires extremely long system operating life and is divided into phases ranging from long periods of moderate activity to very short periods of high activity. The reliability and performance requirements vary with these mission phases. The system operating environment places demanding weight and power constraints on the system and subjects it to thermal and mechanical stress and radiation. The part of the system that this reuse scenario addresses is an algorithm that makes an optimal assignment of space-based weapons to multiple hostile boosters.

The development context into which the reusable asset is to be inserted is based on an iterative design process broadly divided into three phases: baseline determination, initial design, and design refinement. The baseline determination phase determines resource requirements and allocation for the basic architectural and algorithmic structures of the system. The initial design phase consists of trade-off studies to select from the design options being considered. The design refinement phase explores the selected design option(s) to discover and remove any deficiencies in concepts or requirements. This scenario addresses the design refinement phase, assuming that the products of the baseline and initial design phase are available.

During the baseline and initial design phases, a high-level design of an algorithm to cluster targets and to assign and sequence weapons to the target clusters was created. Using this high-level design, a performance analysis was conducted. After this performance analysis, the design was carried to the next level of detail. A second performance analysis determined that an important factor in improving performance was the revision of the design to use of an NlogN sort. This identified a candidate reusable asset. From additional analysis, it was determined that the ratio of processing workload to communication workload was an important factor in how much speedup

was achievable through parallel implementations of the algorithm. This identified matrix computations as an area of interest and provided another candidate reusable asset.

The characteristic attributes for this example are described in Table 7-2. The third column in Table 7-2 indicates the required level of confidence in the certification process for having successfully evaluated the asset with respect to that attribute or for having chosen techniques that are effective given that the asset exhibits that particular attribute. The column labeled "Weight" indicates how important one attribute is relative to all the others.

As this scenario illustrates, the certification process has to be adaptable to a range of domain-dependent certification requirements that are established by a domain analysis that, at a minimum, classifies candidate assets according to criticality and enumerates the characteristic attributes.

This scenario also illustrates that the certification process should also allow the user to consider the impact that different domain assumptions would have made on the certification process and the certification level assigned to an asset. Since a reusable asset will be used in a different system from the one for which it was developed, it will very likely be subject to different requirements than those for which it was certified. Thus, there are potential differences between the certification requirements for a component from the viewpoint of the developer or library and that of the reuser.

These differences can be described by a distance function which specifies the transformations required to adapt the asset from one set of requirements to another. The transformations specify the changes that are necessary to adapt the certification requirements for the asset to the new requirements; the measure of distance establishes objective criteria for evaluating the level of effort required to effect the transformation.

A reuser would like to select a component with the "smallest" measure of distance. A certification process for reusable components should help a reuser assess the certification distance by providing a means of judging the adequacy of the certification process for assuring the level of certification required of the component by the new application. That is, it should allow the reuser to measure the distance between the certification level of the component as it currently exists and the certification level the reuser requires.

7.3 Asset Type

For the element of Asset Type, the asset types listed in Table 7-1 were selected for the CF because they are typical of the types of assets found in the DoD libraries surveyed during the framework development. Asset type is, of course, the major factor in determining the types of evaluation methods that can be used as well as the quality factor(s) relevant to certification. To address the certification needs of a reuse library or repository, a certification framework has to be able to address the full range of asset

types in the library. Thus, the overall schema for defining certification defects, selecting techniques, and judging the effectiveness of methods for detecting defects has to be applicable to different asset types. If not, the consistency of the certification process cannot be maintained across the library.

Table 7-2. Example of domain certification considerations for space application

Domain Specific Attribute Concern	Descriptor	Required Level of Confidence	Relative Weight
Dependability	Reliability	High	Important
Processing	Real-Time, High Workload	Moderate	Less Important
System/Hardware Architectural Impact	High	Moderate	Important
Complexity	Moderate Complexity	Extremely High	Important
Development Formality	Informal Development	Moderate	Important
Software Category	Real-Time	Moderate	Important
Error Detection	Presence of Residual Errors Not Acceptable	Moderate	Less Important
Test Comprehensiveness	Detect All Possible Errors	High	Important
Problem Reports	No Problem Reports Exist	Moderate	Important
Usage Attribute	Known to Have Been Used, but No Data on Usage	High	Important

7.4 Quality Factor

The quality factor is the specific requirements concern against which an asset is being evaluated. As such, it determines the defects or non-conformances that could be exhibited by an asset as shown in Figure 7-1. Not all quality factors are relevant to all asset types, and, furthermore, a quality factor that is relevant to a particular asset type in one domain may not be relevant in another domain.

Also, a particular *subset* of relevant quality factors could be selected for certification based on the business model of the organization responsible for certification. The quality factors for the certification framework, previously listed in Table 7-1, were selected from the Rome Laboratory Software Quality Framework (SQF) [BOW85] [SPS95], the Guide for Information Technology on asset certification developed by the Reuse Library Interoperability Group (RIG) [COM96], and the software characteristics of the ISO/IEC 9126 [INT91]. The certification framework has to be able to differentiate among the quality factors in defining a certification process while maintaining the consistency of the certification approach.

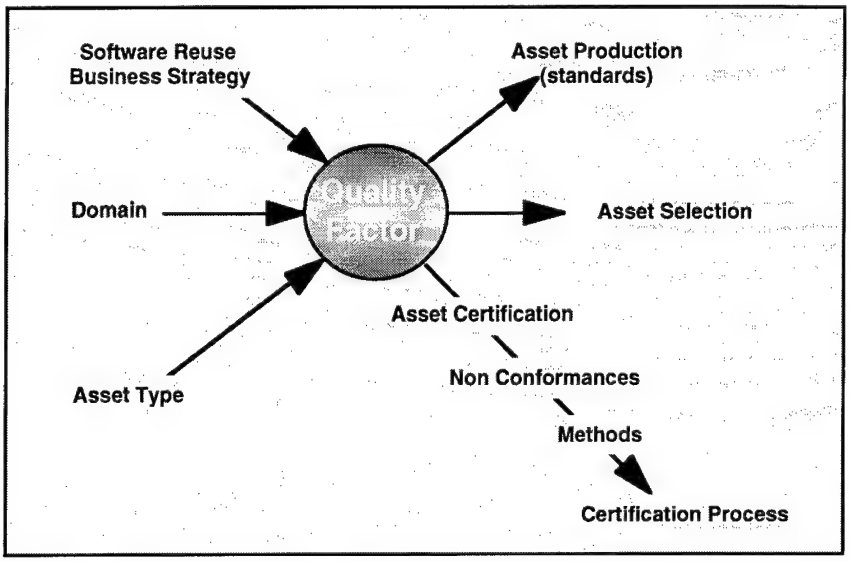


Figure 7-1. Quality factors influence the resulting certification process

7.5 A Thread through the CF

The CF encompasses a broad view of the reuse context, yet a thread can be constructed through the CF by selecting specific attributes within each reuse context element. Selecting attributes for each of the elements provides the user a mechanism to apply the CF to his specific situation. Then, the CF serves as a algorithm for decision-making based on the user-defined attributes. Using specific attributes for each of the vertical elements, a particular thread through the tabular view of the CF can be defined as shown in Table 7-3.

Since most organizations may not have all the information available to exercise our certification algorithm, we defined a default profile, based on empirical data collected from studies of industry projects. Our default profile can be used to "get started" and may be fine-tuned with organizational data, as it becomes available. As illustrated in Figure 7-2, our default profile for our certification field trial was optimized for a software reuse business strategy of "Public Library," a domain of "MIS," an asset type of "Code," and a quality factor of "Correctness."

Table 7-3. Thread through the tabular view of the CF

S/W Reuse Business Model	Domain	Asset Type	Quality Factor	Non-Conformance Class	Certification Techniques	Certification Process
Vendor Owned Domain	MIS	Design Information	Correctness	Latent	Compilation	Process Definition
Gov't Supported Standard	Avionics	Document	Completeness	Robustness	Static Analysis	Procedures
Value-Added Reseller	C^2	Test Artifacts	Understandability		Inspection	Tools
Gov't Owned Architecture	Automated Test Equip.	Req. Specs	Performance		Testing	Data Collection
Gov't Owned Domain	Weapon Systems	Code	Fault Tolerance		Formal Verification	Certification Levels
Reengineering	Communication	Architecture	Functionality		Benchmarking	
Public Library	Intelligence	Database Schema	Maintainability		Modeling	
Commercial Library	Process Controls	Models	Portability		•	
	•	Video	Reliability		•	
	•	•	Usability		•	
	•	•	Safety		Other	
	Other	•	Security			
		Other	Availability			
			Testability			
			Survivability			

A discussion of the elements of "Non-Conformance," "Techniques," and "Process Elements," as well as guidance for selection of other "Quality Factors" and associated tools and techniques, is found in the Volume 2, the Certification Framework. Additional details about the certification field trial are found in Volume 5.

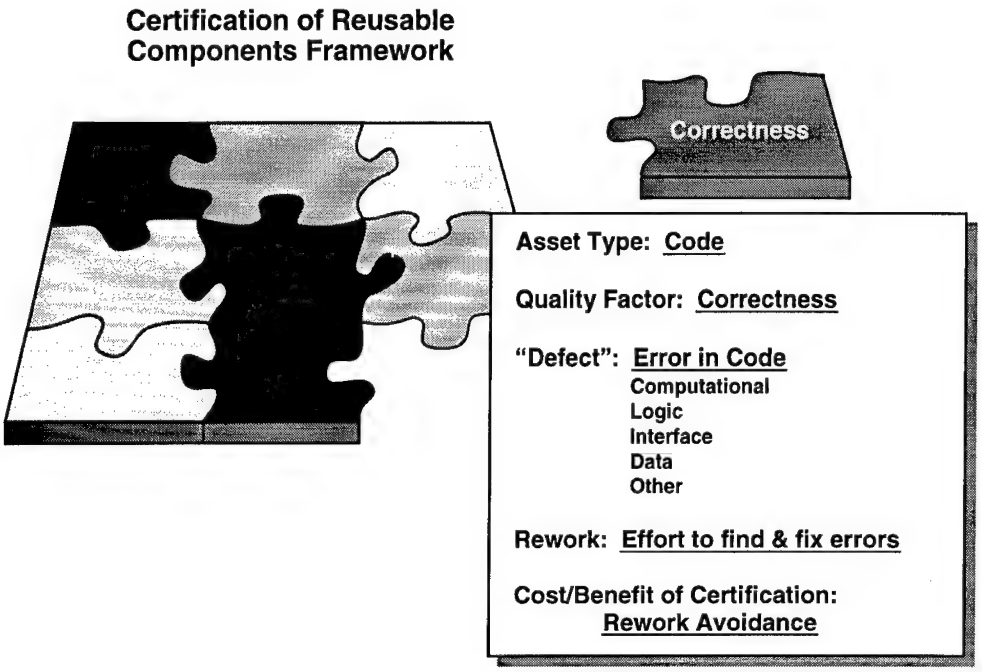


Figure 7-2. CF tailored to the quality factor of "Correctness" for "Code" assets

8 Project History

The highlights of CRC's thirty-month project history are best captured by the time line illustrated in Figure 8-1.

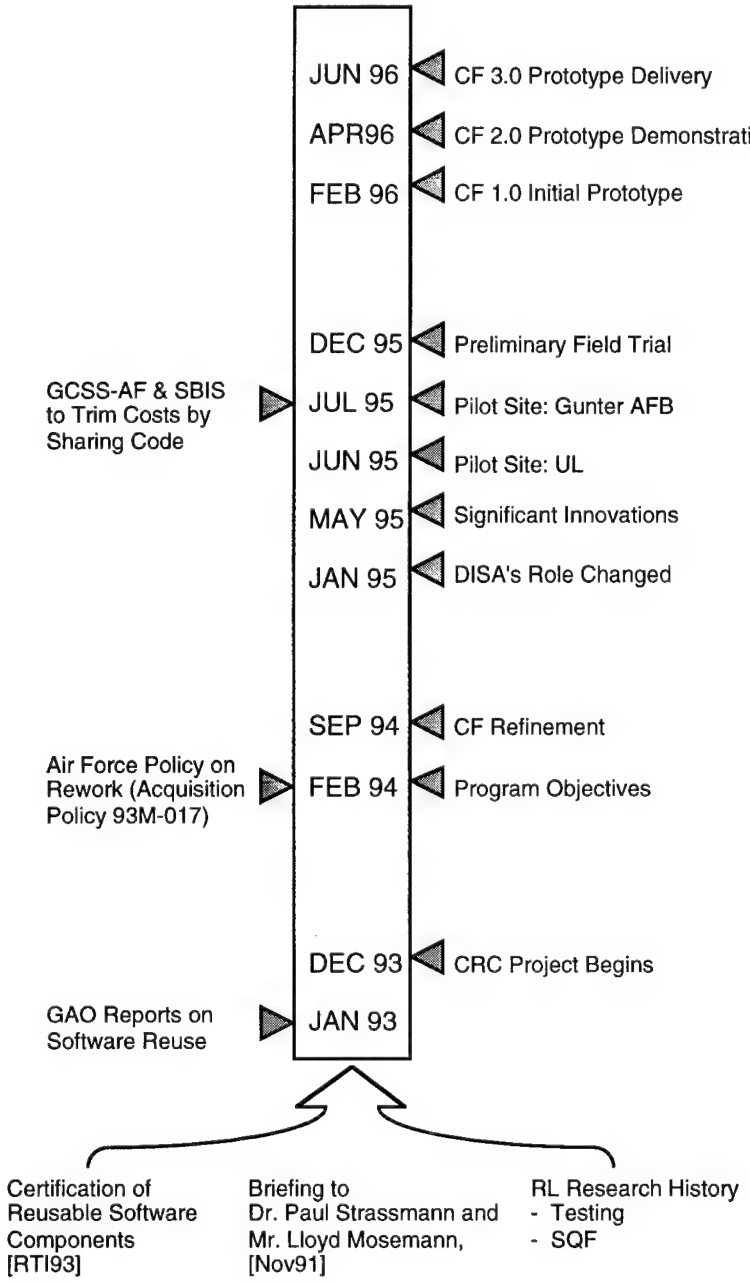


Figure 8-1. CRC chronological project history

As shown on the time line during January 1995, DISA's role changed from our initial plan, and their staff members were no longer able to participate and provide a pilot site

to apply the CF. The CRC Team searched for a new pilot site, and in July 1995, successfully recruited Gunter Annex of the Maxwell AFB and the Global Combat Support System-Air Force (GCSS-AF) program as a pilot site.[1] This activity required significant time and resources, yet was critical to the success of project's transition of developed technology. Despite these obstacles, the CRC Team was able to add another potential pilot site at UL. The CF will be applied and validated at these sites under separate funding.

Through the life of the project, the CF was developed as an evolvable technology as required by the CRC Statement of Work. As shown on the time line in Figure 8-1, the development of the CF 1.0 began at the start of the project and this initial version of the CF was completed in February of 1996. Review of the Cost/Benefit Plan and the operational concept, together with the results of the initial field trial, led to expansion of the CF 1.0 to include scope and rigor, resulting in CF 2.0. Additional development refined CF 2.0 leading to the final CF 3.0. This approach of the CF as an evolving, iterative technology is best represented in Figure 8-2.

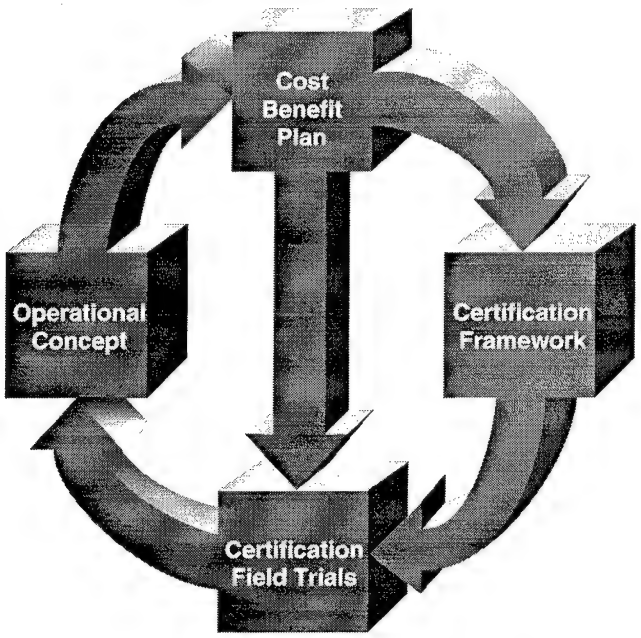


Figure 8-2. Evolution of the CF

Major CRC work products of the project were a final CF 3.0, a Cost/Benefit Plan, and OCD, a study of a Certification Toolset, and a Code Defect Model. The CF was applied in a field trial using the selected certification toolset and guided by the certification process, procedures, and data collection forms. The results of the field trial were

[1] GCSS-AF was previously known as the Base Level Systems Modernization (BLSM) Phase II program.

analyzed and documented in Volume 5 - Certification Field Trial. Additional details about the field trial and each of the other major work products can be found in the appropriate volumes of the CRC supporting documentation suite.

9 Project Results

This section discusses the technical results of certification, the important findings and conclusions for the following areas:

- Certification Framework
- Cost/Benefit Plan
- Operational Concept
- Certification Field Trial
- Certification Toolset
- Code Defect Model

Additional details for each of these task areas are found in the supporting volumes of the CRC documentation suite.

9.1 Summary of Certification Framework

The Certification Framework is a method for deriving certification processes that help assure reusable asset quality, using the most cost-effective means available. The Certification Framework specifies the types of assets that can be certified and the quality attributes for which each type can be certified. The process by which each type of asset is certified for each relevant quality attribute includes the techniques and tools to evaluate the asset as well as precise procedures to be followed.

The quality attributes relevant to a particular asset are often a function of both the domain in which it was developed and the domain in which it is to be reused. Furthermore, the desired level of quality can be affected by the business model under which the repository or library operates because the cost of certifying various levels of quality has to be balanced against the objectives and resources of the library. Thus, the framework has been designed to be adaptable to the various reuse contexts. Quality is a general concept that can be difficult to describe or measure directly. The Certification Framework manages this difficulty by associating quality with a generic notion of non-conformance; the fewer non-conformances an asset has, the higher its quality. All non-conformances are defined by a defect profile that identifies the types and density of defects (non-conformances) an asset may exhibit. Given this defect profile, an appropriate set of certification techniques can be selected based on a certification technique effectiveness profile. This latter profile predicts how effective particular evaluation techniques will be at detecting the defect types. A cost/benefit model was developed to select and sequence these certification techniques to optimally detect defects. The benefit of performing certification is measured by rework avoidance.

Rework was selected in order to assess risk and show the economic value of certification.

The remainder of the discussion in this section presents the base Certification Framework which covers the latent and robustness non-conformance classes. The "Xs" in Table 9-1 illustrates the areas that were researched for the Certification Framework.

Table 9-1. Certification Research Areas

	Non-Conformance Class	
Asset Type	Latent	Robustness
Software Source Code	X	X
Software Architectures	X	--

In order to use this methodology, several steps must be taken to tailor the Certification Framework for a specific situation. The Certification Framework provides a decision support mechanism for constructing a context-sensitive certification process, as illustrated in Figure 9-1. The decision support mechanism is an algorithm that selects non-conformance classes and certification techniques and tools based upon the software reuse business model, the domain characteristics, the asset type, and the relevant quality factor.

First, the reuse context must be identified, including the business model, the domain, the type of asset and the certification quality factor. Secondly, the types of defects for the combination of asset and quality concern for each of the non-conformance classes are determined and the defect density and rework effort associated with each defect type are derived. Given this information, an appropriate set of certification techniques or methods are selected to detect the defined defect types. Finally, the cost/benefit of performing the certification for this scenario is measured by rework avoidance. Table 9-5 illustrates the data required for the cost benefit optimization and Equation M-2 presents the optimization calculations. The result is a certification process which produces a higher quality asset that is more likely to be reused.

The set of selected techniques and their order of application form the core of the certification process comprising an organization's certification policy. In this way, the Certification Framework operates much like cost modeling tools used in planning development projects, but instead is applied to planning certification processes.

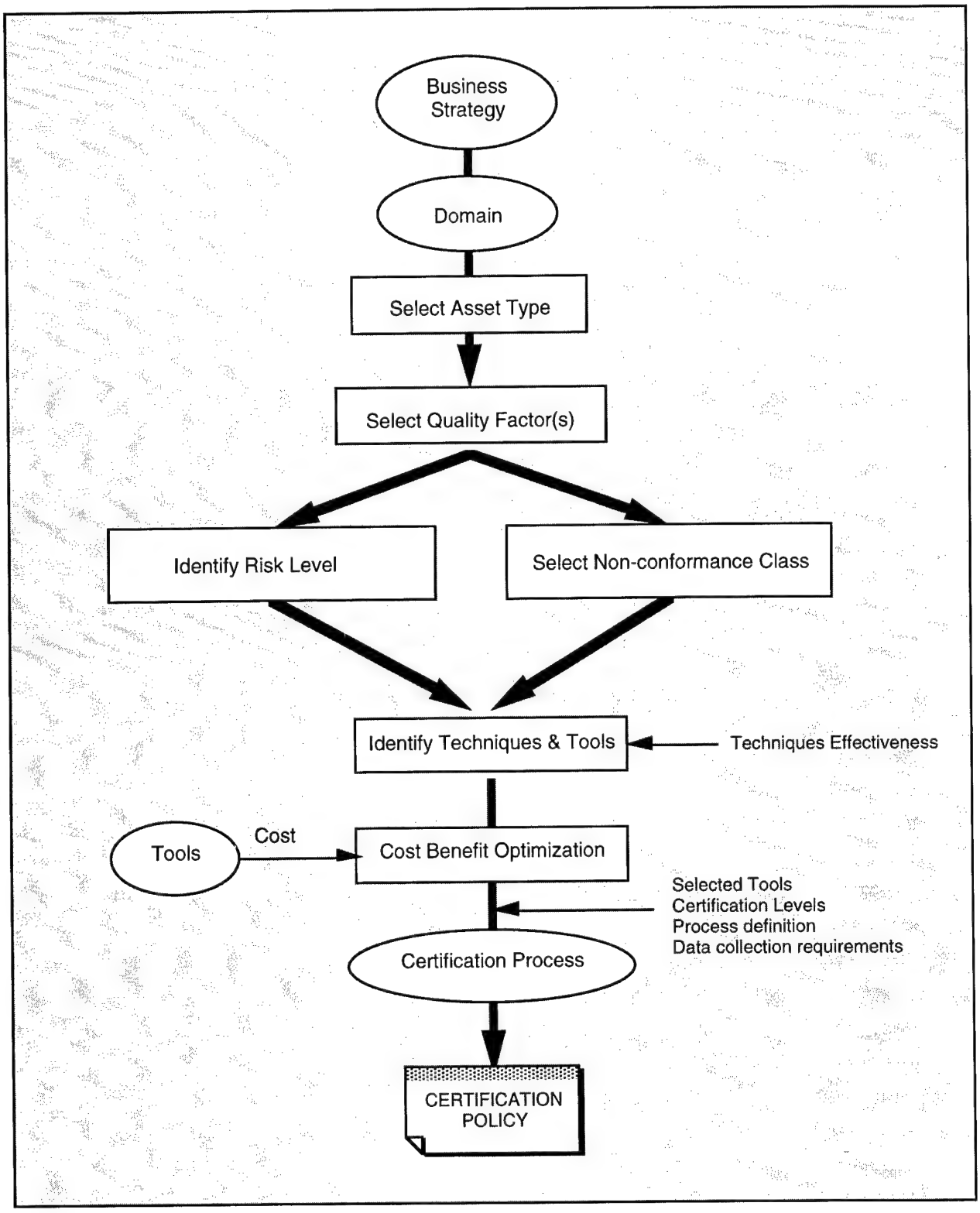


Figure 9-1. Decision support mechanism for the Certification Framework

In order to fully utilized this decision support mechanism, the details of the framework elements must be examined. The Certification Framework is comprised of three types of elements as shown in Figure 9-2:

- reuse context
- defect model
- process

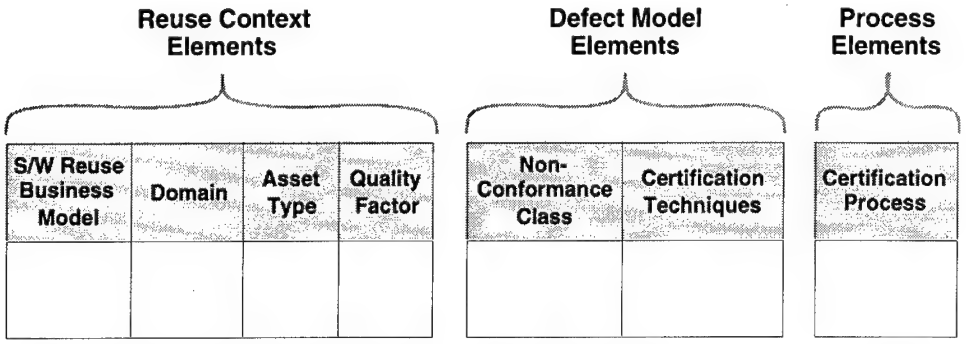


Figure 9-2. Certification Framework element groups

The Certification Framework is used to construct a certification policy, which is a set of certification processes that apply to a specific asset type, and that certify specific quality factors. A certification policy is related to the Certification Framework elements in the following way:

- reuse context elements specify the circumstances under which the policy is applicable
- defect model elements define the certification objectives
- process elements describe how to perform the certification

9.1.1 Defect Model Elements

After the reuse context for certification has been established by identifying the business model, the domain, the asset type, and the quality factor(s), the appropriate defect model is chosen and used to specify the certification process for that context. The defect model is comprised of the elements of non-conformance class and certification techniques as shown in Table 9-2. Non-conformance classes are described by a defect profile comprised of defect density and defect rework. Likewise, certification techniques are described by a technique effectiveness profile consisting of removal effectiveness and cost.

Table 9-2. Defect model

Defect Model Elements			
Non-conformance class		**Certification techniques**	
Defect profile		Technique effectiveness profile	
Density	Rework	Removal effectiveness	Cost

9.1.2 Non-Conformance Class

The Certification Framework defines a model for certification based on quality, where quality is defined according to an absence of defects (or non-conformances) with respect to requirements. This is expressed by a set of quality factors each having a defined defect model. The fewer defects an asset contains, the higher its quality. A hierarchical model of defects, called a non-conformance model, specifies a partitioning of the defects into classes. The base Certification Framework addresses non-conformance classes of latent and robustness. Latent defects are defects relative to an asset's original requirements that were not found and corrected prior to submission of the asset to the reuse library. Robustness defects are defects that only arise when reusing an asset in a different context.

For each non-conformance class, the Certification Framework focuses on developing a detailed understanding of the types of defects that comprise that class and the techniques that can detect those defects. This understanding is captured in a defect data profile which details, for each type of asset and each quality factor, the types of defects, their density, and associated rework. Table 9-3 provides a conceptual view of a defect data profile.

Table 9-3. Defect data profile

Defect Type	Defect Density	Rework Effort
Defect 1	Type 1 Defects/KSLOC	E1
Defect 2	Type 2 Defects/KSLOC	E2
.	.	.
.	.	.
Defect n	Type n Defects/KSLOC	En

A requirements violation of any kind is considered a *defect* if it must be corrected or dealt with before the asset can be reused (i.e., its presence requires rework). Defects are studied in the context of a particular quality factor and type of asset. For example, an organization concerned with certifying the portability of software components would identify and study defects that affect source code portability. Specific examples might include operating system function calls, dependence on a vendor-specific file format, or the use of a compiler-provided program.

In developing the defect data profile, individual defects are grouped into *defect types* for convenience in analysis. Using the example of source code portability, individual defects might be grouped into the following defect types: external interface, numeric representation, and language usage. Different densities, rework costs, and detection techniques are then associated with these different classes.

Density refers to the number of defects per unit size measure (e.g., per thousand lines of code (KSLOC)). An expected density is estimated for each defect type for uncertified assets. Using the example of portability defect types, a 10,000 line program with 43 external interface defects would have a defect density of 4.3/KSLOC for the external interface defect type. The size component used to compute defect density can be any convenient, countable, well understood, and consistently applied measure. Defect density is important in selecting certification techniques, determining the order in which they should be applied, and understanding how they should be applied. Defect density is also used in calculating the value of avoided rework.

Rework, measured in person-hours per defect, includes all effort associated with isolating, fixing, retesting, and documenting a defect, on average. Rework effort can be converted into cost (i.e., dollars) for a particular organization by multiplying by the organization's average labor rate for a certification engineer. Users benefit from certification by being able to select high quality assets and avoid rework. Avoided rework is used to measure risk reduction and place an economic value on certification activities.

9.1.3 Certification Techniques

In addition to the defect data profile, a profile of certification techniques, their effectiveness for detecting each of the defect types, and the costs associated with using these techniques is generated; this is shown in Table 9-4. Effectiveness profiles describe the number and percentage of defects, by type, that different techniques and tools can detect and the costs of applying the techniques and tools. These costs of applying techniques and tools, incremental costs, are measured in person-hours to apply techniques and use tools, as well as investment costs associated with tools (e.g., acquisition, training, and maintenance). This approach takes into account the following two important findings:

- techniques are not equally effective at finding all types of defects
- techniques vary significantly in their cost of application.

These considerations explain why designing a certification process is an exercise in trading off benefits versus costs.

Table 9-4. Techniques effectiveness profile

	Defect Removal Yield for Techniques/Tools					
Defect Type	**Tool A**	**Tool B**	**Tool .**	**Tool .**	**Tool .**	**ToolN**
Defect 1	%1A	%1B	.	.	.	%1N
Defect 2	%2A	%2B	.	.	.	%2N
.	.	.				.
.	.	.				.
.	.	.				.
Defect n	%nA	.	.	.	.	%nN
Investment Cost	IA	IB	.	.	.	IN
Incremental Cost	CA	CB	.	.	.	CN

Like the Nuclear Regulatory Commission's *Guidelines for Verification and Validation of Expert System Software and Conventional Software* [MIL95], the Certification Framework incorporates an approach known as Fault-Specific Verification (FSV), where the choice of certification methods (i.e., techniques) is based on the types of faults (i.e., defects) that can occur. Once the defects are identified, the set of methods is chosen based on the asset type, the methods' effectiveness in identifying the specific defects and the degree of rigor indicated by the risk class. This is the basis for the Certification Framework defect models.

The degree of rigor is not only a function of which certification techniques are applied, but also of the acceptance or exit criteria for each technique. The same technique may be applicable to more than one risk class, but for the higher risk class, it may have more stringent acceptance criteria.

The optimal situation is the scenario of unlimited certification budgets and all techniques and tools could be used. Each technique would act as a filter in a pipeline through which an asset was passed. Each technique application would remove the

defects from the asset that it was equipped to detect and the asset would emerge at the end of the pipeline purged of all defects. This scenario is represented in Figure 9-3. However, a more realistic situation is the one where resources are constrained and the certifier must choose which techniques and tools he can afford to implement. To facilitate this trade-off process, a cost/benefit model was developed as a key feature of the Certification Framework. This model assists the certifier in identifying the most cost-effective subset and ordering of certification techniques within his budget constraints.

The Certification Framework cost/benefit model combines the data from the defect data profile and the techniques data profile with the cost data as shown in Table 9-5 in order to create a tailored certification process.

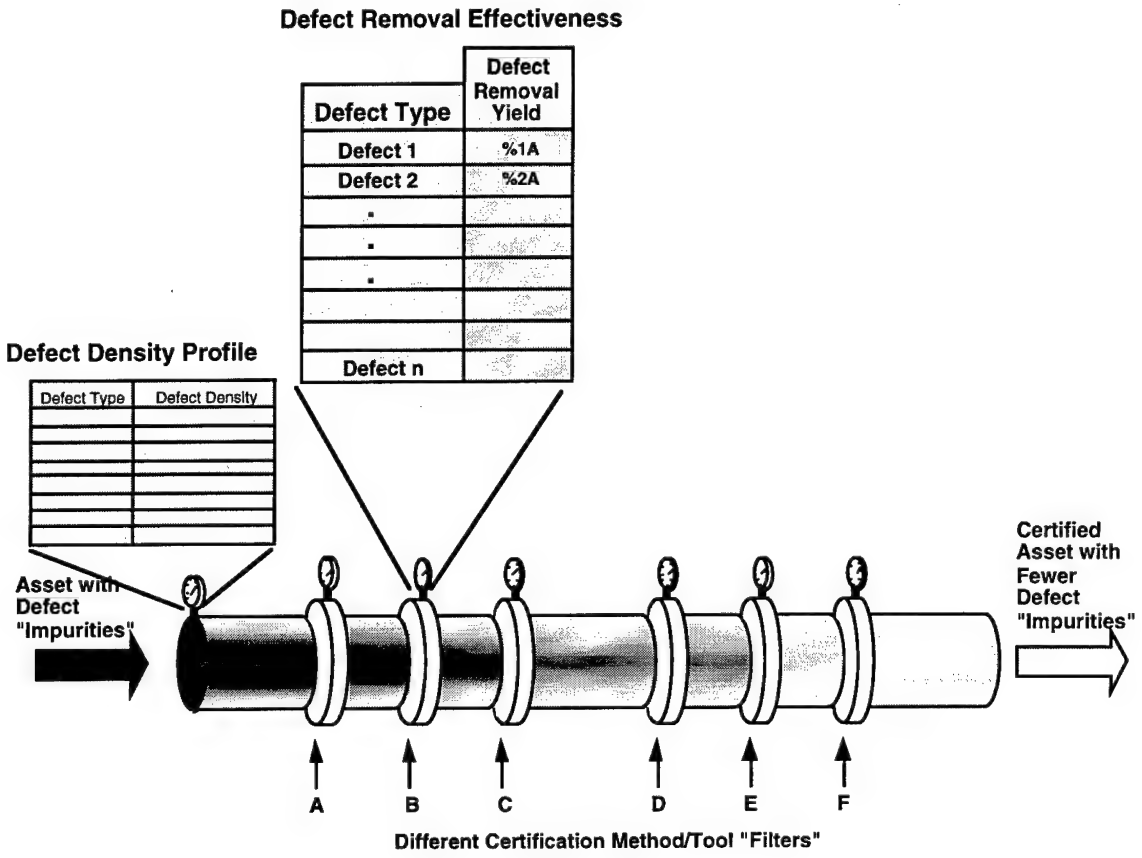


Figure 9-3. Certification method filters and effectiveness

Table 9-5. Cost/benefit optimization data elements

	Certification Method 1		...	Certification Method m	
Defect Type	**Defect Removal**	**Rework**	...	**Defect Removal**	**Rework**
Defect 1	%Yield11	$A11/KSLOC	...	%Yield1m	$A1m/KSLOC
Defect 2	%Yield21	$A21/KSLOC	...	%Yield2m	$A2m/KSLOC
...	...	...	...	...	...
Defect n	%Yieldn1	$An1/KSLOC	...	%Yieldnm	$Anm/KSLOC
Investment Cost	$I1		...	$Im	
Incremental Cost	$D1/KSLOC		...	$Dm/KSLOC	

Ideally, the reuse organization would use historical information, such as software problem report data, to develop the data for the profiles and cost model. However, industry averages for the defect densities can be used until the organization is able to institute its own data collection program.

After all of the profile data has been compiled, the Certification Framework cost/benefit model can be applied to optimize the certification activities.

9.1.4 Process Definition

Every certification process can be located within a 3-dimensional structure, shown in Figure 9-4, where every process P on the cube represents a particular certification process to certify an asset type to a given level against a particular non-conformance class for a particular quality factor. Certification levels may be based on expected rework and/or other measures of risk. The non-conformance class, quality factor, and certification level that determine P are also influenced by the corresponding software reuse business model, the domain, and the asset type. This family of certification processes constitutes an organization's certification policy.

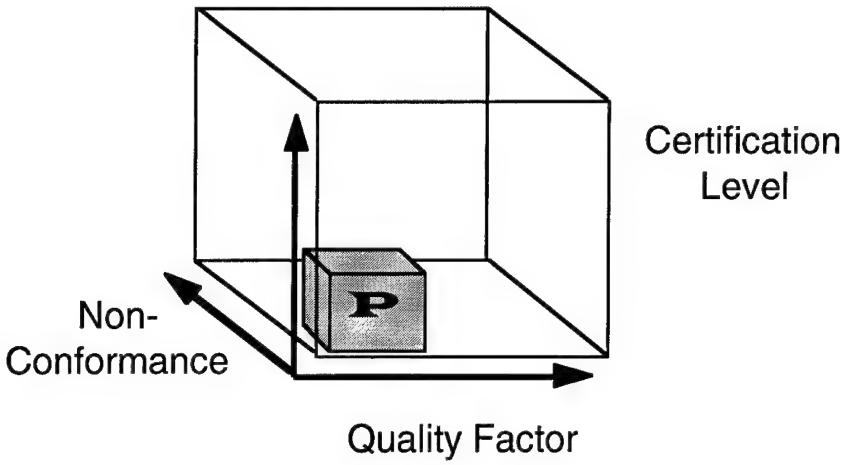


Figure 9-4. Family of certification processes for an asset type

The Certification Framework provides a method for deriving certification processes that help assure reusable asset quality, using the most cost-effective means available to an organization. Figure 9-5 illustrates the overall operational context of the Certification Framework within an organization.

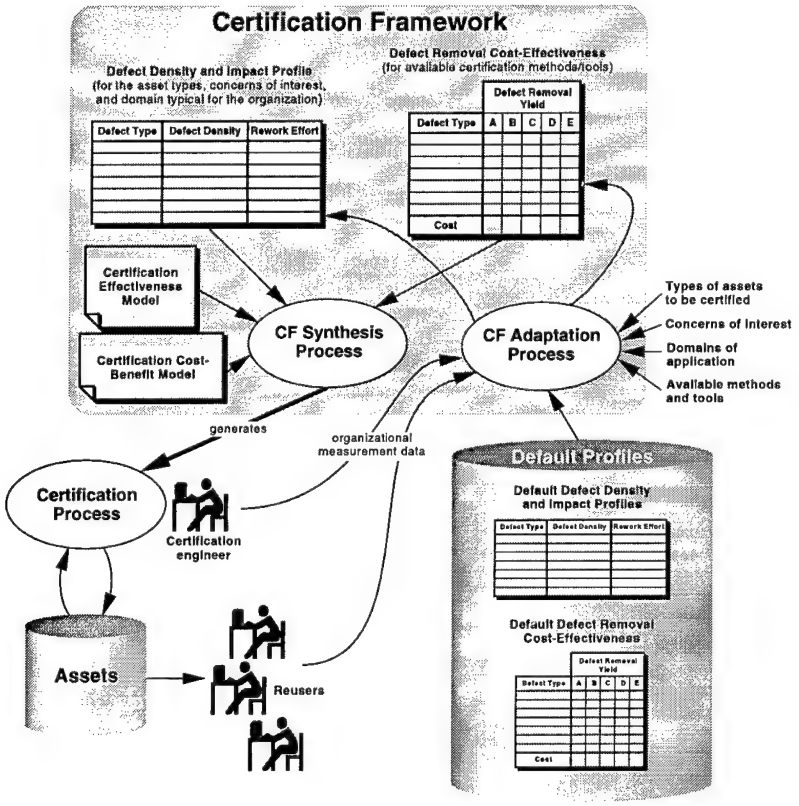


Figure 9-5. Certification Framework operational overview

The process that results from the activities in Figure 9-5 is that which the certification engineer follows to certify an asset. The process techniques are identified through the cost/benefit optimization. This certification process definition also includes the specific steps and procedures that should be followed during certification activities.

9.1.5 Certification Levels

The base Certification Framework defines multiple certification levels based on two aspects: non-conformance class and the risk level. The following section presents the details of selecting a risk level for certification.

The need for certification, and the degree of certification, depends on the risk classification of the system in which the reusable component will be used. The Certification Framework has a three-level risk classification adopted from the verification and validation (V&V) risk classes defined in the Nuclear Regulatory Commission's (NRC)*Guidelines for Verification and Validation of Expert System Software and Conventional Software* [MIL95]. The NRC's classification scheme was designed to encompass high-reliability, safety-critical systems. Risk Class 1 is the highest risk level and Class 3 is the lowest; therefore Class 1 requires the most stringent certification. A primary concern for Class 3 is minimizing the consequences of poor quality for the project (i.e., rework). On the other hand, the primary concern for Class 1 is damage to people or the environment.

Determining which risk class a system falls into is a function of two aspects: degree of required system integrity, and system control/complexity. The degree of required system integrity is based on the acceptability of the consequences of system failure or incorrect operation. The economic, legal, environmental, ethical, and business consequences listed in Table 9-6 are considered.

Table 9-6. Consequences Considered for Determining Degree of System Integrity [MIL95]

Consequences of System Failure or Incorrect Operation
injury or death to plants and animals
interruption of system service
financial loss
loss of information
inconvenience to people
destruction or pollution of the environment or ecosystem
disruption of the system's mission
impact of the availability or operation of other systems
loss of opportunity
impact on an organization's capability to perform
loss of human lives
human injuries
long-term health problems
discomfort to people

The degree of system integrity required is determined by Table 9-7 below.

Table 9-7. Determining Required Degree of System Integrity [MIL95]

Degree of System Integrity	Acceptability of Consequences (Table 3-9)
High	Unacceptable from any perspective
Medium	Somewhere between High and Low
Low	Reasonably acceptable from all perspectives

System control/complexity characteristics are summarized in Table 9-8.

Table 9-8. Determining System Control/Complexity [MIL95]

System Control/Complexity	System Characteristics
Quite High	System is a safety system, or is specifically designed to support or relate to a safety system System directly controls, or provides real-time advice to an operator to control something
Moderately High	System involves at-real-time or near real-time processing or any of the following: • distributed processing • embedded processing • complex reasoning • interrupt-driven processing • a large number of complex interacting systems
Low to Moderate	System is basically a stand-alone user-driven consulting system

Once both degree of required system integrity and system control/complexity are known, the risk class is determined from the decision table shown in Table 9-9. The next table, Table 9-10, provides examples of the different risk classes taken from the Nuclear Power industry. Similar analysis must be done by any organization responsible for setting certification policy.

Table 9-9. Risk Classes (1 = Highest Risk)

	Degree of Required System Integrity		
System Control/Complexity	**Low**	**Medium**	**High**
Quite High	Risk Class 2	Risk Class 2	Risk Class 1
Moderately High	Risk Class 2	Risk Class 2	Risk Class 2
Low to Moderate	Risk Class 3	Risk Class 3	Risk Class 2

Table 9-10. Examples of Risk Classes from Nuclear Power Industry [MIL95]

System Control / Complexity	Degree of Required System Integrity		
	Low	**Medium**	**High**
Quite High Embedded Real-time Continuous Data-Input channels Direct Control Functions Many have Interrupt Processing	**Risk Class 2** Steam Generator Blowdown Control System Radioactive Waste Management	**Risk Class 2** Automatic Control-Rod Manipulation Main Feedwater Control System	**Risk Class 1** Reactor Protection System
Moderately High Embedded or Attached No Direct Control functions Control-Decision Support functions At least near real-time Continuous Data-Input Channels	**Risk Class 2** Thermal Plant Analyzer (TPA) Turbine Generator Diagnostic Monitoring	**Risk Class 2** Emergency Operating Procedure Tracking System (EOPTS) Reactor Safety Assessment System (RSAS) Reactor Emergency Action Level Monitor (REALM)	**Risk Class 2** Emergency Core Cooling System (ECCS) Real-time Monitoring and Diagnosis
Low to Moderate Stand-Alone User-Driven Non Real-time Advisory Functions No continuous Data Input	**Risk Class 3** Fuel-Rod Reshuffling Planner Water Chemistry Advisor	**Risk Class 3** Safety Review Advisor (SARA) Plant-Layout	**Risk Class 2** In-service ECCS Inspection Advisor Emergency Safety Actuation System (ESAS) Testing System

9.1.6 The Economics of Certification

For the lowest level of risk, Risk Class 3, the cost of certification can be weighed against the cost of not performing certification—the cost of rework. The certification process either discovers defects or confirms their absence, and then, the benefit of certification is avoidance of rework on the part of the consumer of the reusable asset. Selection of certification techniques is governed by the cost/benefit model described in CRC Volume 3, Cost/Benefit Plan.

For the higher levels of risk, Risk Class 2 and Risk Class 1, the cost of failure is great. The cost of failure in these cases is difficult to quantify and it may overshadow the cost of rework. The objective is to determine which certification techniques achieve the required level of risk reduction at the lowest total cost. However, other types of risk besides rework must be considered.

Responsibility for the costs of certification is a function of the software reuse business strategy, and may largely depend on who bears the risk of failure. The business strategy dictates whether certification should be performed by the producer of reusable assets, the consumer of reusable assets, or an independent third party. For example, it is difficult to imagine a business strategy in which a public reuse library would provide personnel or funding to certify to Risk Class 1. Given the large investment in certification and the risk of liability for Risk Class 1, it is more likely that the cost would be allocated to the development costs of a new system, or would be recouped by commercial sale or licensing of the certified assets.

From the point of view of the consumer of reusable assets, reuse makes sense if the cost of reusing an asset is less than the cost to develop it from scratch. Many software development cost models assume that reused code or modified reused code costs significantly less to develop than new code. In this scenario, certification can be substituted for V&V that would have been performed if the asset were developed from scratch. Therefore, from the consumer's point of view, the level of certification must meet or exceed the V&V requirements of his development environment. If there is a shortfall, the consumer is responsible for the cost difference; thus, he is motivated to acquire assets certified to the highest available risk class. Even if the available assets are not certified to the risk class he needs, the certification still represents a V&V cost savings.

9.1.7 Certification Framework Synopsis

Table 9-11 summarizes how the defect model and process elements of the base Certification Framework are related to the two aspects of certification levels: non-conformance class and risk class. In general, the certification process for each risk class incorporates all of the techniques applicable at the next lowest risk class, plus includes additional techniques and/or requires more stringent exit criteria.

Table 9-11. Certification Levels

Non-Conformance Class	Latent	Robustness
Description	Defects remaining after development	Internal defects wrt. use in new context
Quality Factor **Defect Types**	Quality Factor$_{1,1}$ Defect Type$_1$ Defect Type$_2$ ... Defect Type$_n$	Quality Factor$_{2,1}$ Defect Type$_1$ Defect Type$_2$ ... Defect Type$_n$
	Quality Factor$_{1,2}$ Defect Type$_1$ Defect Type$_2$ ... Defect Type$_n$	Quality Factor$_{2,2}$ Defect Type$_1$ Defect Type$_2$ ... Defect Type$_n$
	...	...
	Quality Factor$_{1,n}$ Defect Type$_1$ Defect Type$_2$ ... Defect Type$_n$	Quality Factor$_{2,1}$ Defect Type$_1$ Defect Type$_2$ ... Defect Type$_n$
Risk Class	**Certification Process**	
3	Technique$_a$ +... + Technique$_{a+n}$	Technique$_a$ +...Technique$_{a+n}$
2	Level 3 plus Technique$_b$ +... + Technique$_{b+n}$	Level 3 plus Technique$_b$ +... + Technique$_{b+n}$
1	Level 2 plus Technique$_c$ +... + Technique$_{c+n}$	Level 2 plus Technique$_c$ +... + Technique$_{c+n}$

Interviews with potential users have revealed that certification is perceived as a labor-intensive but required activity. Uncertain of the benefits, and with little or no way to quantify the benefits, most organizations are trying to reduce the cost of current certification activities. This situation is not too surprising given the general state of practice in software testing and quality assurance. Testing and quality assurance activities can consume a significant portion of software development resources, often as much as 40% or more of the total project budget. Yet many organizations do not have a basis for measuring the benefit of their testing and assurance efforts or improving the results. All too often the approach is to "bang on the code" as much as possible with the time and staff available.

The Certification Framework directly addresses the problem of quantifying the cost and benefit of certification by providing the following:

- A basis for quantifying, understanding, and comparing the costs and benefits of using different certification techniques and tools,

- A method for deriving certification processes for different types of assets, with different quality concerns, in different application domains, and
- A method for predicting the cumulative cost and benefit associated with applying recommended certification techniques and tools.

These capabilities are critical in achieving a major benefit of certification: enabling reusers to select low risk reusable assets and thereby avoid the costs associated with reworking low quality assets or developing new ones. Using the Certification Framework can help organizations answer questions like the following:

- How can different qualities or asset characteristics be certified? How are these qualities represented in our assets?
- What kinds of defects can we expect in our assets? What qualities do these defects affect? How expensive are these defects to find and fix?
- Which techniques and tools should we use? Which are the most cost-effective? What kinds of investments would yield the greatest benefit?
- How efficient and accurate are existing certification processes? How can they be improved to yield greater benefit for the same or lower cost?
- Given a limited budget, what certification activities can and should be performed? Who should perform these activities?
- Are process improvements contributing to higher quality? Is certification worth doing?

The application of the Certification Framework has demonstrated that it can detect defects in components and that the key to the effectiveness of the framework is the targeting of techniques to particular types of defects. This ability to target techniques to defect types provides an underlying rationale for developing certification processes. Moreover, it maximizes the effectiveness of the process while minimizing its cost. For this approach to work successfully, however, the organizations that use this framework must give careful consideration to the types of defects and the relative distribution of those types in the assets they are certifying. More research into the strengths and weaknesses of evaluation techniques for different defects is also needed. Although the focus of this effort has been on reusable components, the approach has potential for application to the broader area of software V&V. Thus we believe that the payoff from such research would be lower software V&V costs and higher levels of assurance in the quality of software.

While the Certification Framework addresses the problem of how to confidently assess the quality of reusable software components, it does not completely address who should conduct the required activities. Alternative approaches include all evaluations conducted by the certifying agency, all evaluations conducted by the developer and

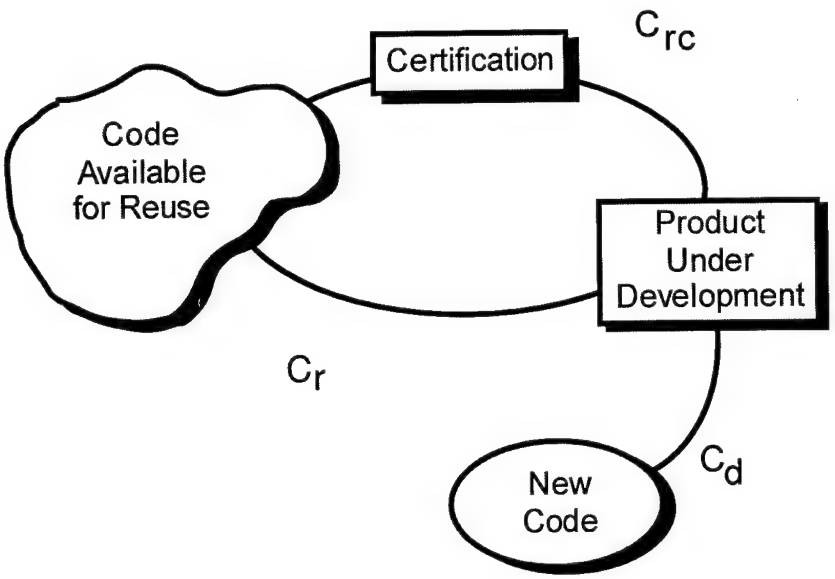
reviewed for compliance by the certifying agency, and a mixture of certifying agency and developer evaluations. This question could be addressed by evolving the framework into a standard for certification. Developers could then incorporate the verification activities of the standard into their development process, and certifiers could certify that a component had been developed according to the standard.

Finally, the Certification Framework recognizes the importance that architecture plays in the analysis and reuse of software components. It does this in two ways: (1) by using architectural analysis to assess the quality of software code components and (2) by providing guidelines for certifying architectures as reusable components. Architectural analyses are important in understanding how components "work" and evaluating how they "fit" with other components. Continued development of this field of study could be the key to effective reuse in the development of verifiable complex systems.

Additional details about the CF can be found in the supporting document titled CRC Volume 2 - Certification Framework.

9.2 Summary of Cost/Benefit Plan

In developing a cost/benefit plan for the certification of reusable assets it is essential to distinguish between the costs and benefits of reuse in general, and the costs and benefits of certification. Figure 9-6 illustrates the problem. The producer of a software system is envisioned to have three sources from which a software asset may be obtained: 1) it can be developed new at a cost of C_d, 2) it can be reused without regard to certification at a cost of C_r, or 3) it can be reused with the benefit of certification at a cost of C_{rc}. One way of quantifying benefits is to compute cost avoidance. The cost avoidance of reuse is the difference C_d-C_r. The cost avoidance of certification is C_r-C_{rc}. Other benefits besides cost avoidance are possible, but are outside the scope of the cost/benefit model proposed in this plan.



Benefit of Reuse in General = $C_d - C_r$

Incremental Benefit of Certification = $C_r - C_{rc}$

Figure 9-6. Context of Cost/Benefit Models

This plan focuses on determining the marginal cost/benefit effect of adding certification to an existing reuse program. Unfortunately, an extensive survey of published cost/benefit models related to reuse identified only one model that specifically addressed certification issues. That model suffered from several important limitations: it provided inadequate resolution of investment and operations cost of certification, did not quantify the benefits of certification, and did not consider the effect of certification on an asset base. The lack of an appropriate pre-existing model made it necessary to develop a new cost/benefit model targeted at the effects of certification on reuse.

The first step in developing a cost/benefit model is to identify the costs and benefits to be captured. The costs of certification include the following:

- cost of acquiring certification tools, training, etc. (investment cost)
- cost of executing the certification process for an asset (incremental cost).

The normal costs of operating a reuse program, such as those associated with a repository are not specifically considered in the certification cost/benefit model.

The certification of reusable assets offers two benefits distinct from those of reuse in general. These are as follows:

- reduction of risk in reuse

- increase in attractiveness of reuse.

Certification does not, by itself, improve the quality of an asset. The risk of reuse can be characterized by three major factors of the asset: correctness, understandability, and completeness. The premise of certification is that if resources are put forward to address or improve each of these factors, the probability or risk of defects in the asset will be reduced. In this context, the degree of risk equates with the expected amount of rework encountered by reusing an asset. In order to decrease the degree of risk, you must increase the degree of correctness, or understandability, or completeness, or all of these factors by instituting certification. This will result in a lower average rework level, μ_C, and a decrease in the variability of rework, σ_C, for the set of certified assets. This relationship is illustrated in Figure 9-7. The focus of this cost/benefit plan is to increase the degree of correctness in a cost effective manner in order to achieve a reduction in risk. Rework cost avoidance, *Ca*, will be used as the measure of risk reduction. Therefore, if an accurate certification scheme is applied, reusers are less likely to be surprised by failures and rework. They can select more reliable assets and avoid less reliable assets.

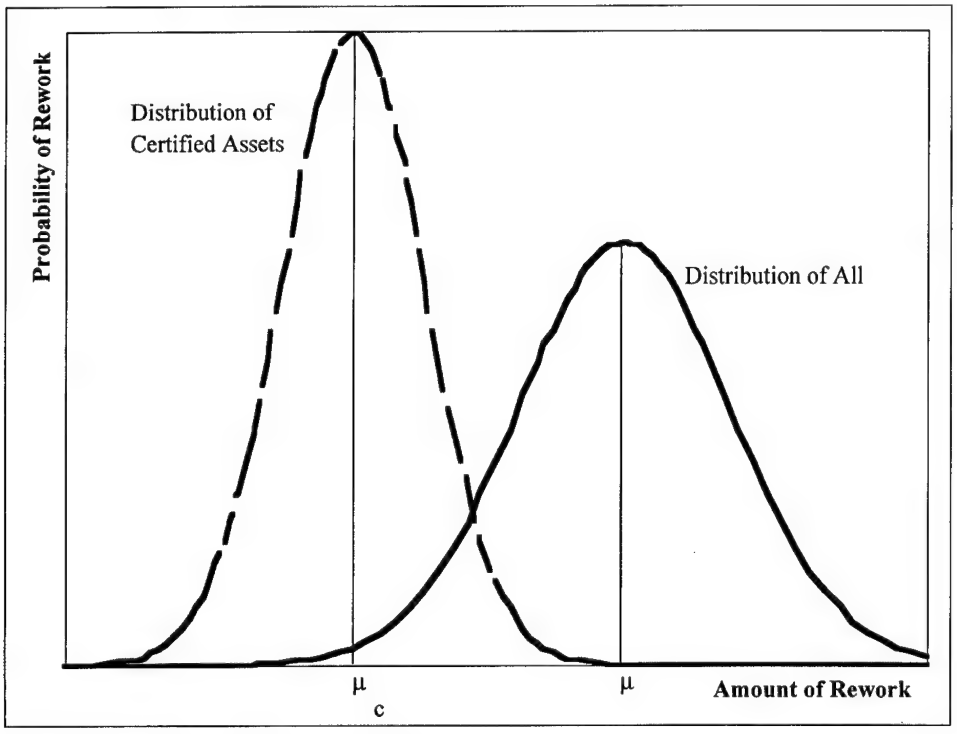


Figure 9-7. Certification Risk Reduction

Concurrently, the increased quality that is psychologically associated with a *certified* asset makes it more attractive to potential reusers. Thus, the level of reuse is likely to increase, in addition to the reduction of risk already described as a benefit of certification. Early results from the DISA Reuse Metrics Program [CHU93] support this hypothesis. Repository

reports showed that more highly certified assets were more likely to be extracted for potential reuse [RAT94].

While risk can readily be converted into a cost avoidance, the marketing benefit of certification is harder to quantify. Consequently, the cost/benefit model proposed in this plan deals only with risk reduction.

9.2.1 Certification Method Effectiveness by Error Type

The type of error or defect and the ability to detect its presence has an impact on the cost of rework. In order to produce a more accurate estimate of cost avoidance, a model must be developed that provides a basis for synthesizing a certification process that provides a measure of the degree to which a software component is free of various types of defects. For the more detailed cost/benefit analysis two aspects must be specified: (1) the type of expected defects and (2) defect detection techniques. In this context, the cost of rework avoided is shown in Equation M-1.

$$Ca_k = \sum_1^m \sum_1^n (D_i \bullet RH_i \bullet LR) \bullet DD_{ij} - \sum_1^m (Inv_j + Inc_j) \qquad \textbf{(M-1)}$$

where

Ca_k = cost avoidance due to certification of asset k

D_i = defect density for defect type i

RH_i = number of rework hours for defect type i

LR = hourly labor rate

DD_{ij} = percent of defect type i detectable by technique j

Inv_j = investment cost for technique j

Inc_i = incremental cost for applying technique j

n = number of defect categories

m = number of certification techniques

An important aspect of Equation M-1 is that it provides the cost of rework avoided when all defect detection techniques are applied in the certification process.

However, due to organizational resource constraints or policies, it may not be possible to exercise all of the techniques. This in turn requires a model to determine the order in which methods should be applied in a certification process to maximize benefit, in terms of reduced risk or rework due to defects, and to minimize cost. This stepwise approach that

maximizes rework avoidance with respect to technique defect detection effectiveness, investment cost, and incremental cost is represented by:

$$\max \quad Ca_k = \sum_1^m \sum_1^n (D_i \bullet RH_i \bullet LR) \bullet DD_{ij} - \sum_1^m (Inv_j + Inc_j) \qquad \textbf{(M-2)}$$

$$\text{w.r.t} \qquad \sum_1^m (Inv_j + Inc_j) \leq B$$

where

B =budget for certification activities

This stepwise approach is similar to the methodology employed in a stepwise regression algorithm. This stepwise certification cost effectiveness algorithm, shown in Equation M-2, is used to calculate the costs and benefits associated with defect detection methods and the order in which the methods should be applied; this is based upon the greatest benefit received, rework avoided, for the cost incurred, investment cost plus incremental cost. This is done by a stepwise analysis of cost-effectiveness, where the method with the greatest marginal cost effectiveness is selected at each step. This analysis continues until the best subset of methods has been selected for which total cost ≤ total benefit. The result is a certification process that is optimized for a specific organization's requirements.

9.2.2 Evaluation of Models and Data Collection

In addition to evaluating certification technology, the data collected per this plan should be used to improve the cost/benefit models and data collection methods defined here. Some of the issues to be considered in this phase of analysis include the validity of the modeling approach and the efficiency of the data collection methods. Figure 9-8 below illustrates the process for Reuse Certification Cost Model validation; the references to reuse cost should be interpreted as reuse certification costs.

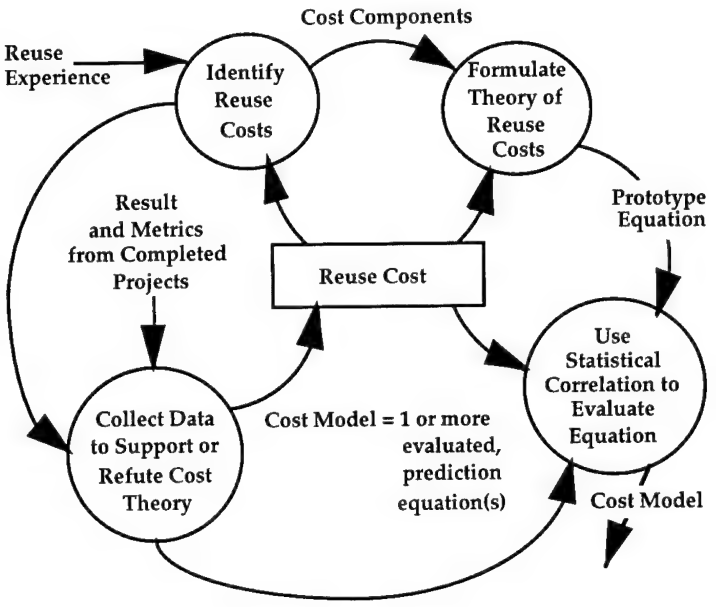


Figure 9-8. Reuse Certification Cost Model Construction Process

Identify Reuse Costs and Benefits: Individual components of cost and benefit were identified. By predicting the cost of the components instead of the whole, the model will take advantage of certain statistics of aggregation. This results in the partial cancellation of errors in predicting the cost of one of the components by opposite errors in predicting the cost of another of the components. The statistics of aggregation work for estimating reuse costs to the extent that the component costs are not all subject to increase or decrease for the same reasons, and to the extent that inaccuracy introduced by the decomposition itself does not outweigh the benefits of the aggregation. Decomposition of the reuse components will initially result in cost factors for evident components (for example: Software Cost = Specification Cost + Design Cost + Implementation Cost . . .) and then model these elements separately.

Data Collection: Support or refutation of the theoretical relationship between an element of cost and some measurable factor requires collecting data from a sample of projects or from records of past projects. In order to assure maximum validity, most data to support the cost theory will have to be collected from new projects.

Formulate Theory of Reuse Costs: Statistical analysis, the most commonly applied analysis technique for cost predictor models, must follow, not precede, formulation of a theory of cause. When there is a causal relationship between a cost prediction and an early observable predictor, there is normally a strong statistical correlation between the two. The converse is not necessarily true. Statistical correlation does not necessarily imply cause due to chance correlation between unrelated data items, therefore, it is essential to formulate a theory of cause with which to identify chance correlations

Correlation: Simple linear regression will be used to fit a curve of the basic shape hypothesized by the theory into the set of data points collected. The best-fit curve is defined by actual numerical values for each of the factors in the prototype equation. The resulting equation implies perfect fit, but the actual data points can be expected to be scattered about the prediction line defined by the equation. The amount of scatter will be expressed as a standard error of estimation and presented as part of the equation (i.e., EQUATION ± 10%). The resultant cost model will consist of a set of evaluated prediction equations useful for projecting a particular component of the total cost with a known margin of error. The degree of accuracy obtained is an important consideration. Boehm describes a good model as one that yields an estimate with an 80% likelihood of being within 20% of the actual.

9.2.3 Cost/Benefit Synopsis

The Cost/Benefit Plan describes a systematic approach to evaluating the costs and benefits of applying certification technology within a reuse program. The plan focuses on the benefits of certification in terms of risk reduction; it quantifies the risk reduction effect in terms of cost avoidance.

The cost model for certification is based on the type of error (or defect) and the ability to detect its presence as it impacts the cost of rework. Since resources usually prohibit exercising all possible defect detection techniques, the model determines the order in which methods should be applied in a certification process to maximize benefit, in terms of reduced risk or rework due to defects.

Our approach maximizes rework avoidance with respect to a technique's defect detection effectiveness, investment cost, and incremental cost. Our stepwise certification cost effectiveness algorithm is used not only to calculate the costs and benefits associated with defect detection methods, but also the order in which the methods are applied. The result is a certification algorithm that can be optimized for a specific organization's requirements.

Since most organizations may not have all the information available to exercise our certification algorithm, we defined a default profile, based on empirical data collected from studies of industry projects. Our default profile can be used to "get started" and can be fine-tuned with organizational data, as it becomes available. Our default profile is optimized for the quality factor of "Correctness" and the component type of "Code." If organizations are interested in other quality factors, the CF provides guidance on the selection of other techniques and tools.

The Cost/Benefit Plan presents a systematic approach to evaluating the cost/benefit of certification technology, in general, and the CRC team's proposed certification framework, in particular. The overall approach includes defining formal cost/benefit models, collecting the corresponding data from the cooperating repository(s), and implementing a comprehensive program of analysis.

The technical approach presented in this plan also helps to mitigate the program risks identified by the CRC team. Some of these risks and the corresponding mitigation strategy incorporated in this plan are as follows:

Risk	Strategies
Certification benefits not quantifiable or not large enough to detect	Consider qualitative effects of certification; conduct more sensitive field studies
Certification not sufficiently automatable to be cost-effective	Develop a generic approach that can apply to any certification criteria allowing flexibility to change
Inability to collect sufficient data from cooperating repository(s)	Conduct field studies; use industry data to parameterize the cost/benefit models

The methodologies and results presented in this study indicate that the current state of reuse practice - many assets of low or indeterminate quality - could benefit significantly from effective certification technology. The implementation of this plan provides information essential to designers of certification programs and operators of reuse repositories, while at the same time minimizing program risks.

Additional details about evaluating the tradeoffs between certification's costs and benefits can be found in the supporting document titled CRC Volume 3 - Cost/Benefit Plan.

9.3 Summary of OCD

The operational concept of the Automated Certification Environment (ACE) can be illustrated as shown in Figure 9-9. A Component Certifier certifies components according to an "instantiated" Certification Framework. The chosen instantiation is driven by the particular needs and issues that can be addressed in reuse and certification.

The Component Reuser searches the repository for candidate components. Once identified, he evaluates the certified component and determines if he can reuse the available component. His decision making is based on information available to him and other users (Cost Analysts, Managers, Data Analysts, Independent Certification Organizations) from the ACE as determined by his selected concern(s). Repository Organizations accept components from Component Suppliers to catalogue and maintain information about each component. Component Creators provide components to Component Suppliers. The Certification Framework Providers establish the instantiated Certification Framework for the user along with training and consultation services.

The users of the ACE are Component Creators, Component Suppliers, Component Reusers, Cost Analysts, Repository Organizations, Component Certifiers, Managers, Data Analysts Independent Certification Organizations and CF Technology Providers.

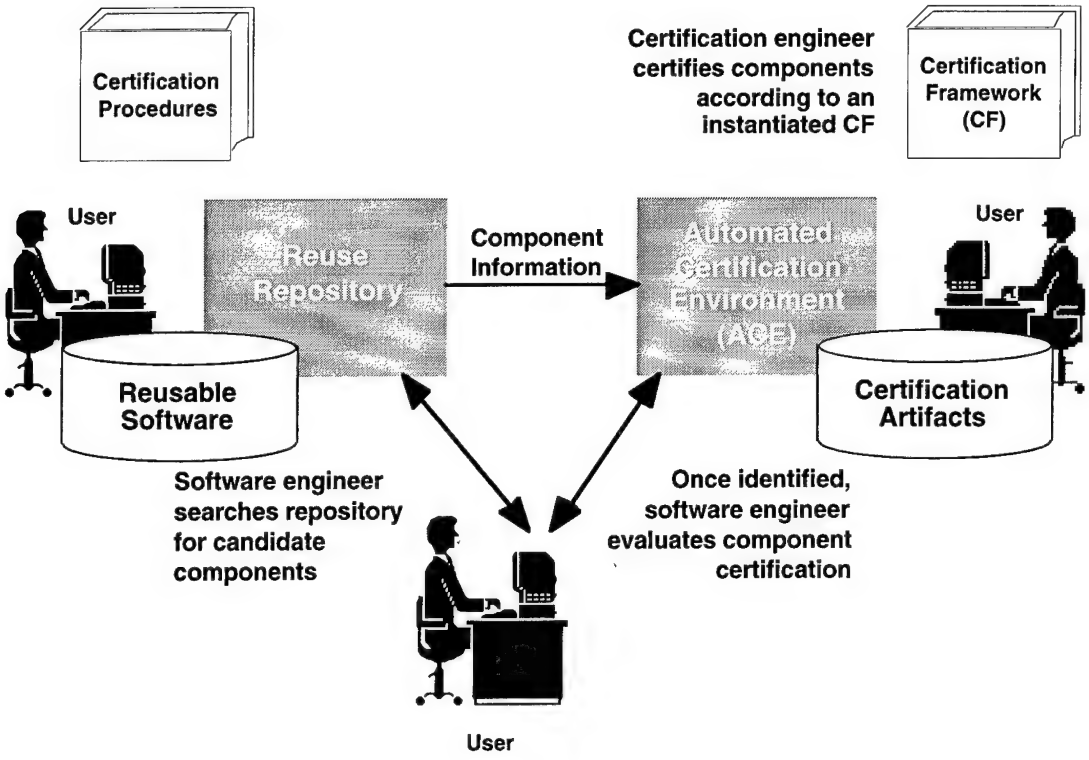


Figure 9-9. ACE operational concept

Representative user scenarios were developed illustrate how users employ the ACE capabilities to accomplish the various activities associated with certification. Scenarios for the ACE can be envisioned within two enterprise settings; a Reuse Library Organization and a general Software Development Organization. As illustrated in Figure 9-10, a Reuse Library Organization may be a Government Repository or a Contractor Repository. A Software Development Organization may be a member in the Commercial Reuse Industry.

In both scenarios, the data needed to drive the CF is part of the overall measurement program of the enterprise. Within each enterprise, a Certification Analyst is responsible for building and maintaining default profiles for different domain application areas and different asset types. The output from the CF is one of many inputs to the creation of the certification policy for each enterprise.

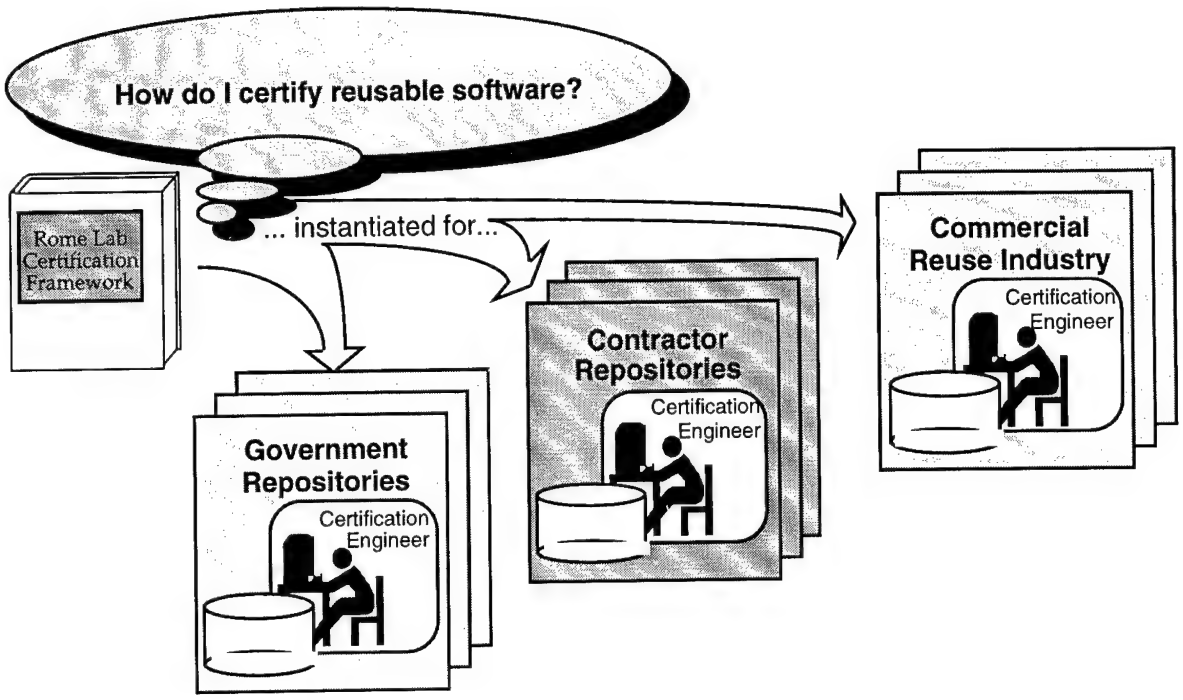


Figure 9-10. ACE scenarios of use

The Certification Analysts, as "players" in these scenarios, may consist of the following kinds of users:

- Librarians and catalog administrators responsible for the quality of an asset collection
- Software developers seeking to provide reusable assets
- Software developers seeking to reuse existing assets
- Test and quality assurance engineers involved in certifying and validating the quality of software

In the usage scenario at a Reuse Library Organization, Government Repositories supply assets for external reuse while Contractor Repositories create and supply reusable assets for their own internal use. This scenario addresses these questions:

- Is it is cost-effective to raise the certification level of my asset?
- Is it worthwhile to continue testing my asset for a specific area of concern?
- Shall I reuse and modify a certified asset or construct a new one for my specific domain and application?

By focusing on defects, the CF and the Cost/Benefit Model provide a straightforward way to quantify the cost of these activities and compare it to the benefit gained.

This scenario describes a representative application of the ACE for certifying a reusable software component. First, the component is assessed and assigned a certification level as defined by the Reuse Library Organization. Areas of concern are identified and the associated methods, techniques and tools are chosen to address the desired areas of concern. A profile of the asset's defects and predicted density is determined across parameters of the kinds of defects (Computational, Data, Interface, Logic, Other).

The cost impacts of measuring each kind of defect is determined using values of such as defect distribution, rework hours, rework cost and rework density. A profile of certification methods/techniques and their predicted effectiveness in detecting defects is determined. After applying all the necessary methods and techniques, a final value of "rework avoided" is computed. Then, a total cost of investing in all methods and techniques is tallied. To complete the scenario, the user is automatically provided with data about his asset to assist in his analysis, and finally, make his decision.

This scenario demonstrates the ACE's features of profiling defects, predicting defect density and cost impacts across a range of defect types. The following events outline the "plot" of this scenario:

1. The user identifies the kind of asset to be certified (source code, architecture, etc.)

2. He picks a user concern(s).

3. He picks a default set of tools and data that is compatible with his current operating environment.

4. He acquires the CF and the Cost Benefit Model.

5. He provides his own data, if available, to the ACE.

6. He applies the Cost/Benefit Model.

7. He collects the computed data.

8. He makes a economic decision based on technical data.

9. He performs defect detection.

10. He customizes, tailors, and updates his CF and his cost model. Each customization may result in different certification solutions.

The CF can be used in this scenario to make project-specific decisions about certifying one or more assets. This scenario can be in the absence of a certification or to replace and/or enhance an existing policy.

The scenario at a Software Development Organization takes place at enterprises that independently certify software and at enterprises that create software (with or without the objective of reuse). This scenario addresses these questions:

- How do I measure compliance with standards in the setting of the certification laboratory?
- Does this asset meet the mandated standards and requirements for the system in which it resides (e.g., consumer safety of goods, transportation safety, flight safety avionics, high reliability of spacecraft, environmental constraints, etc.)?
- How confident am I with the laboratory measurements, estimates and predictions?

Even though these issues are complex, a wealth of theoretical and empirical data about different software measurement and testing techniques exists and can be synthesized to assemble appropriate certification procedures.

This scenario describes a representative application of the ACE for certifying a software component. The component may be part of a system upon which stringent requirements are levied; the component must meet these requirements, otherwise, the system fails approval (i.e., Underwriters Laboratories). Failures must be strictly documented so that components can be redesigned, reworked and resubmitted for certification.

This scenario demonstrates the ACE's features of integrating a process, standards, a CF and a Cost/Benefit Model to establish a desired *level of confidence* for a minimum level of required *reliability*. The problem is analogous to filtering impurities out of a fluid in pipe, as shown in Figure 9-11.

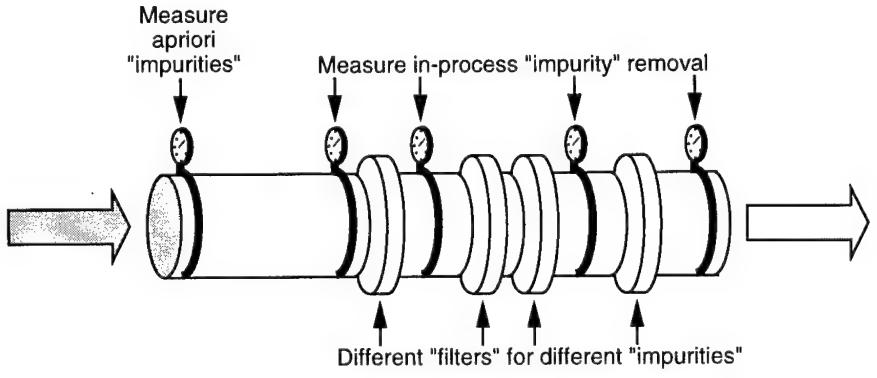


Figure 9-11. Desired level of confidence with a minimum level of required reliability

The first step is *a priori* measurement of the fluid to gain a sense for the nature and concentration of the impurities. If one cannot precisely measure the impurities, one looks to predictive indicators or perhaps considers the source of the fluid or the process that produced it. Based on this *prediction* of the initial impurities, a set of custom filters and measurement devices are assembled to achieve a desired level of confidence in a minimum level of purity at the completion of the process. Different filters address different types of impurities. Measurements *in-process* may further guide or adapt the process.

The following events outline the "plot" of this scenario:

1. The user analyzes high quality components to determine predictive product and process attributes that can be used to determine the a priori characterization of faults.

2. The user analyzes empirical fault detection data for various testing methods and tools to determine the fault detection profiles of different certification methods.

3. The user defines the certification toolset requirements as those collections of measurement (i.e., fault "prediction") and testing (i.e., fault "filtering") elements that can be cost-effectively applied in different combinations to meet different certification requirements of the candidate domains.

4. The user evaluates available certification tools against the requirements to identify a set of candidate tools for further evaluation.

The CF can be used in this scenario to make product-specific decisions about certification compliance.

Additional details about the operational concept can be found in the supporting document titled CRC Volume 4 - Operational Concept Document.

9.4 Summary of the Certification Toolset

This study of the certification toolset accomplished the following:

- Defined certification tool requirements based on empirical data and identified tool capabilities providing the greatest certification benefit for the range of candidate domains

- Evaluated tools based on requirements, mapped tool capabilities to requirements and assessed their level of support for requirements

- Selected tools based on evaluation – identifying the tools that most cost-effectively provide the required functionality

The tool selection process is well defined and repeatable and can be used to evaluate and select new tools and technologies as they are introduced. The selection criteria were focused on software certification in the reuse context; however, they can be customized as suggested by Figure 9-12 to accommodate:

- Additional contexts such as development, maintenance, and reengineering
- Differences in user environment (e.g., tools, personnel, charter)
- Differences in certification objectives (e.g., reliability, maintainability)

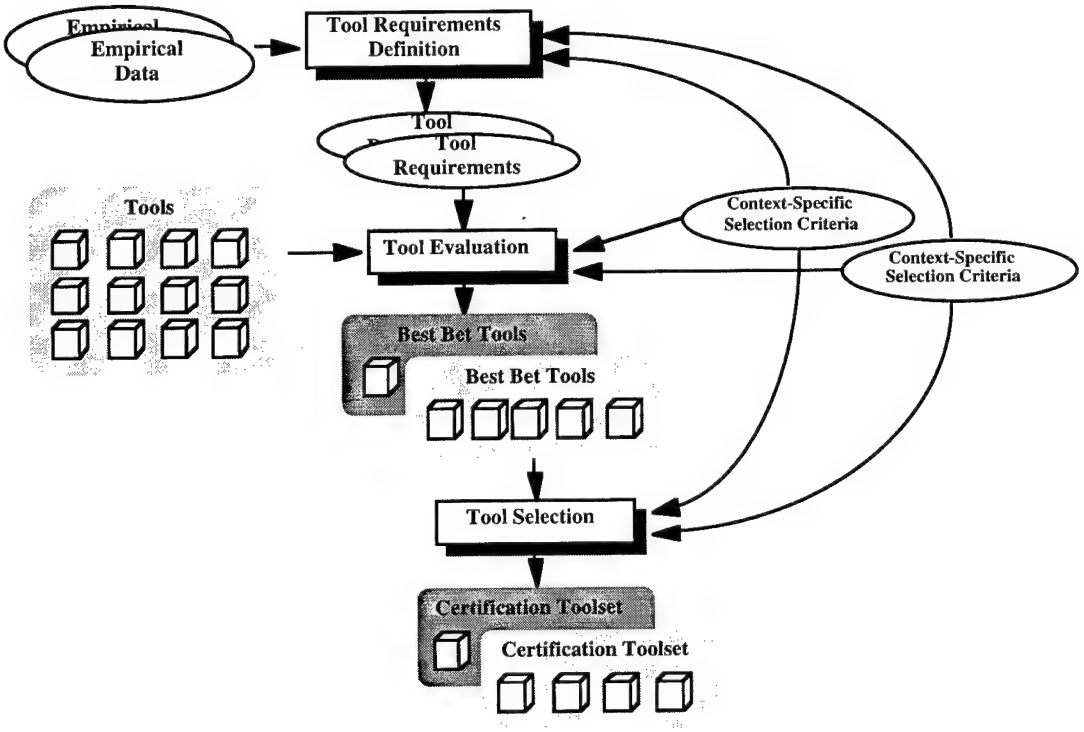


Figure 9-12. Certification tool selection process customization

We derived a "best bet" list of candidate tools that could be effectively used for certification based on the best value in terms of functionality, ease of use, price, performance and integration. From this "best bet" list, the following tool environment was selected for the field trial of an Ada component:

- AdaWise - provides static analysis of alias usage, elaboration order and order dependencies
- Logiscope - provides static and dynamic analysis of control flow diagrams, and structural testing support
- AdaQuest - provides static analysis of style guidelines, size and complexity

The Rational APEX environment supplied the compiler, debugger and code manager while executing on a Sun SPARCstation with the Solaris 2.4 operating system.

Our study of candidate tools for a certification toolset resulted in the following recommendations with respect to Logiscope, AdaCAST, and AdaWise. While Logiscope supports a documented data import capability, it lacks a data export capability. This can probably be worked around by writing filters to strip certification data out of existing tools text reports.

AdaCAST lacks a data export capability. This also can probably be worked around by writing filters to strip certification data out of existing tools text reports.

AdaWise reports identify error locations by source file line number but require the user to manually refer to source code. The output reports could be either modified, or report filters written, to support hypertext traversal to the source code automatically upon termination of the AdaWise program.

With respect to near-term tool requirements, Section 2.1.3 of CRC Volume 6 - Certification Toolset identifies Mutation Testing and Symbolic Execution as technology/technique risks due to their failure to migrate out of the academic and research arena into mainstream software engineering practice. However, the promise and potential value of these techniques to certification warrants additional investigation into ways to facilitate their transition into common use.

Additional details about the certification toolset can be found in the supporting document titled CRC Volume 6 - Certification Toolset.

9.5 Summary of the Certification Field Trial

Given the CRC CF and the Cost/Benefit Plan, we constructed a generic, context-sensitive default certification process. The default certification process consists of four main steps: Readiness Assessment, Static Analysis, Code Inspection and Testing. The default process certifies code components (as opposed to other types of reusable components) and addresses the certification concerns of Completeness, Correctness and Understandability. We developed detailed procedures, data collection forms and guidelines to support the successful execution of the default certification process in our field trials.

The objectives of the certification field trial performed by SPS were as follows:

- Perform all of the steps in the default certification process
- Use all of the tools in the certification tool set
- Assess the accuracy and understandability of the procedures guidance
- Collect effort and technique effectiveness data

- Select a single asset to certify sized for a 2 staff-week certification effort

The field trial was a "hands-on" test of the default certification process as applied to a component. Results of the trial helped assess the accuracy and understandability of the procedures to conduct certification, the effort required to collect data, and the effectiveness of techniques in detecting defects in components.

We initiated the field trial by selecting a component to certify, sized at a two staff-week effort (i.e., employing one Certification Analyst and one Certification Engineer). We selected component #157, the ProGen utility, from the ASSET (Asset Source for Software Engineering Technology) repository distributed on the Walnut Creek Ada CD-ROM.

The ProGen component is 1,543 logical lines of code (Ada semicolons), or 4,387 physical lines of code (non-blank lines). It consists of 10 Ada packages. The component was large enough to not be trivial, and small enough to be certified within a two staff-week effort. ProGen is a utility program that automatically generates prologues for Ada code files. It extracts information such as pragmas, types and representation clauses to construct a prologue. The component includes a main procedure to generate a single executable. It had no recorded defect history and no specification from the component was available.

The Certification Analyst reviewed the ProGen source code by desk-checking and found 2 major defects and 11 minor defects. Therefore, the Certification Analyst seeded 5 additional major defects into the component to provide a significant number of major defects known to her in advance of the field trial. No minor defects were added. The seeded defects were not created in an attempt to duplicate a particular defect profile (i.e., distribution of defect types). The known defects were not shown to the Certification Engineer prior to conducting his tests. While technique effectiveness data was collected, the field trial was not intended to be an experiment to determine the effectiveness of the techniques that comprise the default certification process. The design and implementation of an experiment of that type is quite involved and is significantly beyond the scope of the CRC contract. The effort and technique effectiveness data was collected in order to compare the actual results with comparable values culled from other research studies.

All of our objectives were satisfied by the field trial with the following exceptions. The original test coverage stopping criterion of 100% decision-to-decision path (DDP) coverage was not met for two reasons:

- Logiscope errors led to incorrect display of results in the Logiscope GUI tool.
- It was very difficult to achieve 100% coverage in some units.

One of the originally selected tools, AdaCAST, was not compatible with the Rational Apex Ada environment, and was not used at all in the field trial. The purpose of this tool is to automate creation of test drivers and stubs. The lack of such a tool did not

hamper the field trial because the certified asset contained a main procedure, and test drivers were not needed.

In reference to the asset certified, the resources allocated to the field trial task allowed for certification of a single asset. The asset to certify was selected based on two major considerations: size and defect history. Since the default certification process was derived for Ada code assets, it was understood that the asset must be Ada code. Ideally, the asset should be found in an existing reuse repository.

Size. It was estimated that an asset of about 1000 logical lines of code would be large enough to not be trivial and yet small enough to be certified in a 2 staff-week effort. The effort constraint was developed based on extensive interviews of reuse library personnel performed early in the CRC contract [see CRC's Volume 4 - Operational Concept Document], which indicated that 2 staff-weeks were about the right amount to devote to certifying a single asset.

Defect history. In order to assess the effectiveness of the certification process at finding defects, it was necessary to have an asset with defects known in advance. We were unable to locate suitable Ada code with enough error reports in the configuration managed libraries of in-house Ada development programs. Therefore we selected an asset from the ASSET[2] repository distributed on the Walnut Creek Ada CD-ROM.

Selected Asset

The selected asset was ASSET_A_157, the ProGen utility. This single executable utility program automatically generates comment prologues in Ada code files. It parses the code and extracts information such as pragmas, type and representation clauses used. It had no recorded defect history.

Size of Asset

Logical lines of code	1500 semicolons
Physical lines of code	4300 non-blank lines
Number of packages	approx. 10
Number of files	10 specs, 10 bodies

An informal desk check type code review turned up 2 major defects and 11 minor defects. Therefore we decided to seed in 5 additional major defects in order to have a significant number of major defects known in advance of the field trial. The seeded defects are summarized in the table below. All seeded defects, as well as those found

[2] Asset Source for Software Engineering Technology (ASSET), a division of SAIC. ASSET is now a commercial organization and its assets are available for downloading at the URL http://source.asset.com/WSRD/asset.html.

in the desk check, are documented in Appendix C of CRC Volume 5 - Certification Field Trial and have an identifier starting with "KD_". These known defects were not shown to the certification engineer prior to or during the field trial.

Seeded Defects (all major)

Defect Type	Package Name	Description
Data	ada_scanner	Change Ada reserved word "elsif" to "elseif"
Logic	ada_parser.parse_compilation_unit	Change "if not Done" to "if Done" then exit
Logic	progen_data_structures	insert off-by-one error in for loop
Logic	progen_data_structures	remove reset of counter "Line_Number"
Logic	user_interface	delete loop exit

The seeded defects were not created in an attempt to duplicate a particular defect profile (i.e., distribution of defect types). There are more logic defects than other types simply because these are the easiest type to invent. It turned out to be rather more difficult than we anticipated to create defects that were not caught by the compiler, nor caused immediate catastrophic failure on execution.

Data collection forms described in Section 2.7 of CRC Volume 5 - Certification Field Trial were completed during the field trial. All certification defect reports are in Appendix C of CRC Volume 5 - Certification Field Trial, and the other completed forms are contained in this subsection under the appropriate topic.

Two SPS personnel were involved in the field trial. Their completed Certifier Profile Worksheets follow.

CERTIFIER PROFILE WORKSHEET

CERTIFIER NAME OR ID NUMBER	Joe Tallet
Number of years of programming experience	11
Number of years of programming experience in asset's language	8.5 in Ada
Education (list degrees)	BS/MS Computer Science
Experience with Certification Tools (hours with each tool before starting certification process)	
Rational APEX Environment	2 hrs est.
AdaWise	1 hrs
Logiscope	8 hrs
AdaQuest	1 hrs

CERTIFIER PROFILE WORKSHEET

CERTIFIER NAME OR ID NUMBER	Karen Dyson
Number of years of programming experience	8
Number of years of programming experience in asset's language	4 in Ada
Education (list degrees)	BS Civil Engineering
Experience with Certification Tools (hours with each tool before starting certification process)	
Rational APEX Environment	4 hrs
AdaWise	0.5 hrs
Logiscope	24 hrs
AdaQuest	>40 hrs

Asset Description

The information contained on this worksheet is also discussed in subsection 3.2 of CRC Volume 5 - Certification Field Trial.

ASSET DESCRIPTION WORKSHEET

ASSET NAME	ProGen, Ada Prologue Generation Program
Origin of asset	ASSET Repository ASSET_A_157 with seeded defects
Application domain	software engineering utility
Purpose of asset	automatically generates comment headers (prologues) for Ada code files reports pragmas and representation clauses used
Language	Ada
Number distinct "packages" contained in the asset	10
Physical lines of code (non-blank lines)	4387
Logical lines of code (semicolons)	1543
Age of asset	current date 3/17/89
Version number of asset	1.0
Previous inspection and testing activities	unknown
Additional documentation	short readme file

Effort

Effort to apply the techniques for each step of the certification process was reported on the Overall Process Data Worksheet. Included in the reported effort is the effort to record defects, but not the effort learn how to use the tool. The graph in Figure 9-13 compares the actual effort to apply the techniques to the predicted, or default, effort. Default effort data is taken from CRC's Volume 3 - Cost Benefit Plan.

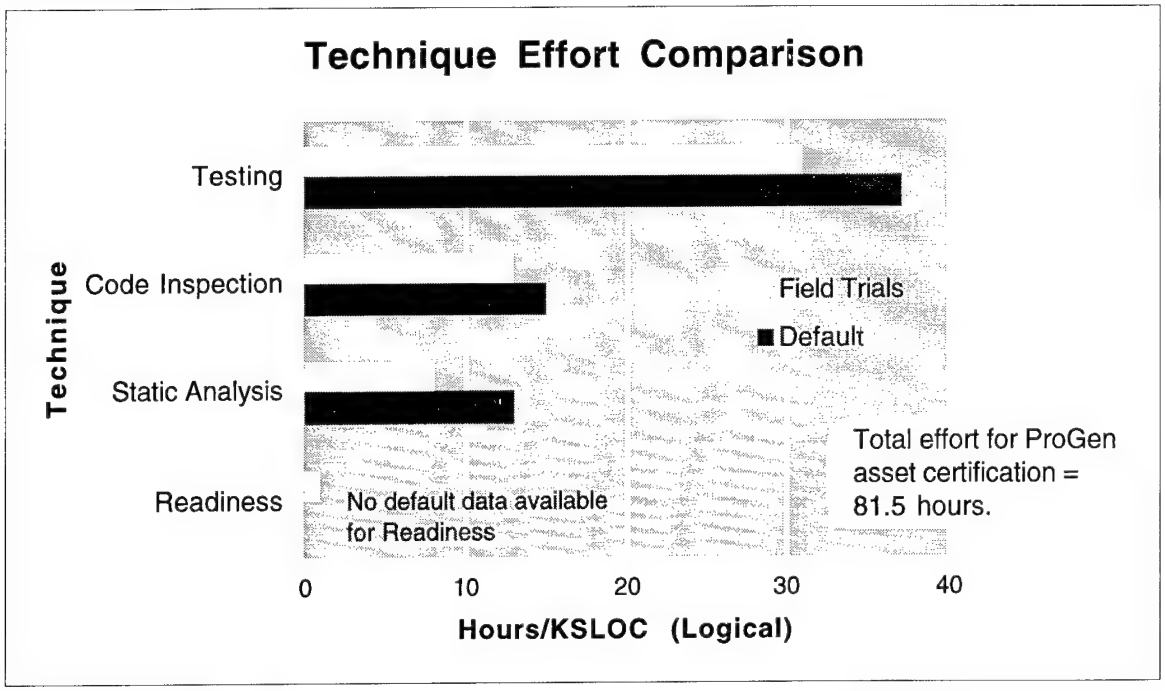


Figure 9-13. Comparison of actual effort to predicted

In general, the actual effort was close to the prediction. However, it must be noted that the testing step was not completed to the point of achieving 100% DDP coverage in all components. It is difficult to estimate how much more effort would be required to achieve this coverage goal.

It seemed to become progressively more difficult to create structural test cases as the coverage increased. This indicates that the effort to achieve additional coverage may have a shape such as is shown in Figure 9-14.

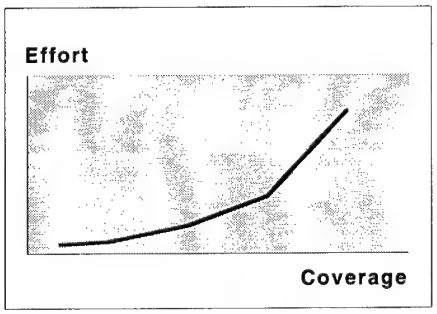


Figure 9-14. Effort to achieve additional test coverage

OVERALL PROCESS DATA WORKSHEET

ASSET: ProGen	Certification Step			
	ASSET READINESS	**STATIC ANALYSIS**	**CODE INSPECTION**	**TESTING**
Certifier ID	Joe	Joe & Karen*	Joe & Karen*	Joe & Karen
Level of Effort (hrs)	1 hr	6.5 hrs Joe 6 hrs Karen	16 hrs est. Joe 4 hrs Karen	8 hrs est. Joe 40 hrs Karen
Problems in Applying Techniques			some questions not appropriate for Ada	can't determine defect category
Problems in Using Tools	Apex: assembler not in path	AdaQuest: ASIS internal error message AdaWise: one analysis "def" does not work; another has ASIS errors Logiscope: insufficient documentation, requires analysis of Ada runtime all: conflicting license manager versions		Logiscope: generated trace file error messages; results inconsistent AdaCAST: not compatible with Apex environment--not used
Problems with Process Guidance			some duplication with AdaQuest audit	
Other Problems				

* for these steps, the Certification Analyst recorded defects on defect report forms

Defects

Many more natural defects were found in the asset during the field trial than were known prior to the start. All are recorded on defect report forms in Appendix C of Volume 5 - Certification Field Trial. Each report has an identifier that indicates the source of the report using the following codes. (No defects were found during the Readiness step.)

Defect Report Identifier Codes

Code	Source
SA	Static Analysis
CI	Code Inspection
TE	Testing
KD	Dyson's Code Review or Seeded Defect

In terms of certification, the asset passed the certification concern of Completeness, and failed in the other two concerns of Correctness and Understandability. In practice, the certifier would face the following choices:

- Reject the asset
- Report the asset as uncertified and record all known defects
- Return the asset to the donor and request repair of known defects; repeat the certification process after repairs
- Repair the defects; repeat the certification process after repairs

Some certifiers may choose to include defect repair as part of their certification process. There is some debate as to whether it would be necessary to repeat the certification process after repairs have been effected, depending on the nature and the number of the defects found. The purpose of repeating the certification would not only be to insure that the defects were repaired, but also to catch any new defects inserted as a result of the repair activity.

Counting Defects. In the following graphs and tables, unless otherwise noted, defects are counted as unique defect reports. The uniqueness criterion means that if the same defect was detected by more than one technique, it is counted only once and credited to the first technique to detect it. In filling out the defect reports, each report is limited to a single package or separately compilable file. All occurrences of the same type of error, such as a style guideline violation, in a package are recorded on the same report, with all defective lines of code noted on the form.

Figure 9-15 shows how many defects were found by the steps in the certification process versus how many are known to exist at completion of the field trial. Defects categorized as *not found* must be, by definition, either seeded defects or those found by informal code review.

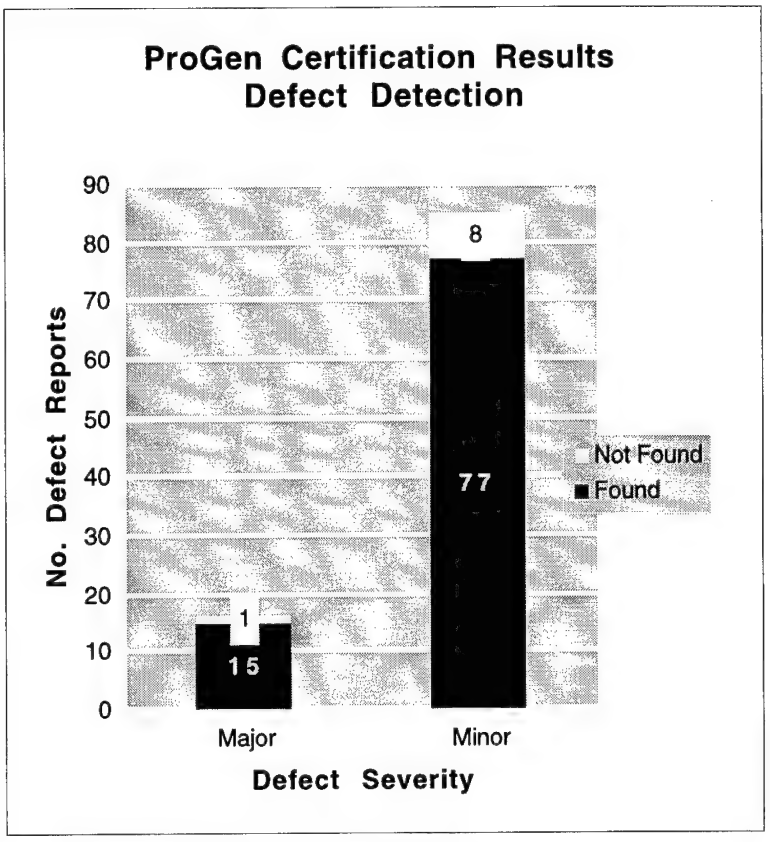


Figure 9-15. Defect detection

Summary of Defect Reports. The following table summarizes the defect reports logged during the certification process steps and the informal code review and seeding activity. Duplicate reports are listed in the "prior step" and "other step" shaded rows.

Defect Report Summary

		Defect Type					
Step	**When Found**	**Comp.**	**Data**	**I/F**	**Logic**	**Other**	**Total**
Readiness	This Step First	-	-	-	-	-	0
Static Analysis	This Step First	0	14	10	24	0	48
Code	This Step First	1	15	13	3	3	35
Inspection	Prior Step	0	0	2	6	0	8
Testing	This Step First	-	-	-	2	-	9
	Prior Step	-	-	-	-	-	0
Seeding &	This Step Only	0	3	2	3	1	9
Review	Other Step	0	3	3	3	0	9

Asset's Defect Profile. Figure 9-16 shows the defect profile of the asset in terms of the known defects. Note that there are seven uncategorized defects that were found during testing. It is important to understand that defects reported during testing are actually failures, and it is not until a failure is debugged that it can be attributed to specific units and lines of code. Debugging was not done as part of the field trial.

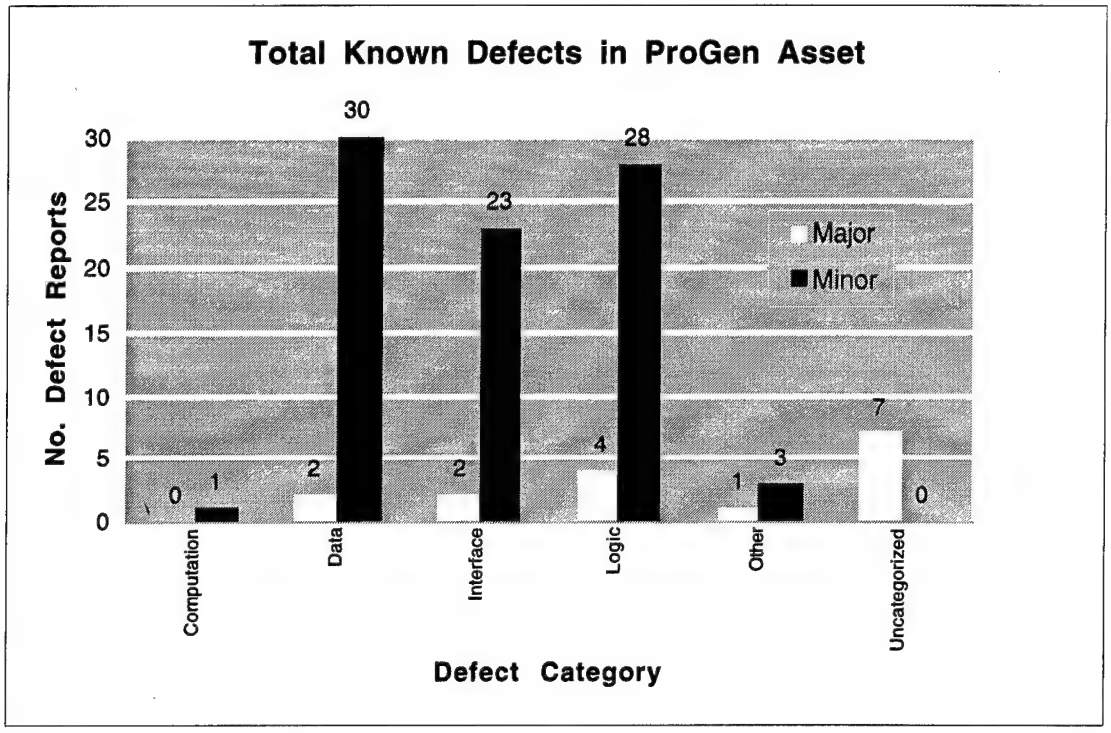


Figure 9-16. Asset's defect profile

The defect density of the asset's major defects, including the seeded defects, is about average for Ada [see CRC Volume 3 - Cost Benefit Plan]. Indeed major defects as we've defined them for the field trial are equivalent to what are typically reported as defects. The number of minor defects was surprising; however, most of these were style guideline violations. The large number of such violations is an indication of the effort that would be needed to take an asset that was not developed subject to these guidelines and make it conform.

Defect Density

Defect Severity	Defect Density (defects/1000 physical lines)	
	Asset's	Average for Ada
Major	4	5
Minor	19	N/A

Figure 9-17 compares the asset's defect profile, including both major and minor, seeded and natural defects, to the default profile [see CRC Volume 3 - Cost Benefit Plan]. One notable difference is that there is a much lower proportion of computational defects. This fact could have two interpretations:

- the techniques used are not effective at finding computational defects
- the asset does not have computational defects

The second explanation is more likely, since the asset is not computational in nature, and one would be hard pressed to find any mathematical expressions in it (other than loop indices). This then indicates that we cannot assess the effectiveness of the techniques at finding computational defects based on the field trial.

In certification, it will typically be the case that an individual asset's defect profile is different from the default profile of any given group of assets. The more that is known about the expected defect profile of assets to be certified, the more cost effective a process can be designed to certify them. For example, if a group of assets to be certified is known not to be computational, then you would not need to include a technique that is effective at detecting computational defects.

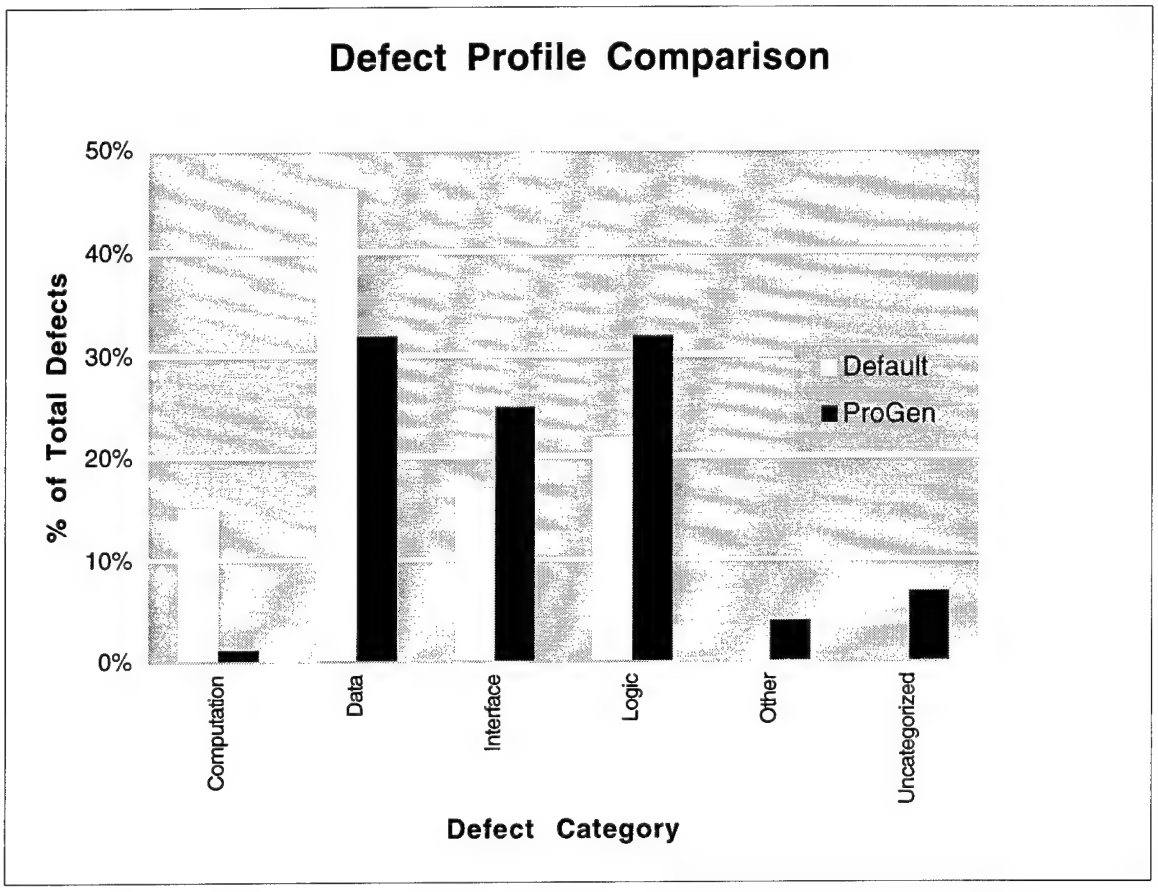


Figure 9-17. Comparison of asset's defect profile to default profile

Technique Effectiveness

As Figure 9-15 shows, all but one of the known major defects was found, and the one not found was a seeded defect. Effectiveness of the default certification process at finding defects is better represented by the proportion of the total seeded defects found than by the proportion of known defects found. This is because there are probably additional natural major defects in the asset, so the total number defects in the asset is unknown.

Effectiveness at Detecting Seeded Defects

Found	Known	Effectiveness
4	5	80%

Figure 9-18 shows the cumulative effectiveness of the steps in the certification process where effectiveness is defined as the proportion of known defects found. From this we can draw several important conclusions. We cannot, however, claim that the combined effectiveness of the default certification process is 90%. As discussed previously in the paragraphs under Asset's Defect Profile, we do not know the total number of defects in

the asset. Furthermore, based on the effectiveness at finding seeded defects, we have reason to believe that more natural defects exist.

Readiness step. There were no defects found during the Readiness step, which means that all code needed to create an executable was available and compiled without error.

The Readiness step is intended to address the certification concern of Completeness. One of the minor defects found during the informal code review (KD_001) was related to Completeness, and it was not found during the field trial. A package specification was included with the asset, but was never withheld by any of the code. In other words, extraneous code was included as part of the asset.

Static Analysis step. As Figure 9-18 shows, only minor defects were found by this step, not major defects. The 55% effectiveness rating for minor defects shown on the graph may be misleading, however. The automated tools used in this step are virtually 100% effective at finding the defects that they are *designed* to find. The effectiveness rating indicates that what the tools are designed to find were only about half of the known minor defects in the asset.

There were no major defects in the asset that are detectable by AdaWise. The type of defects that AdaWise detects are typically designated as major. Thus it is *possible* to find major defects with static analysis tools; it just so happens that there were none in this particular asset. In considering the effectiveness of static analysis in general, is also important to note that many of the major defects that can be found by an Ada compiler would require additional static analysis tools for other languages such as C and C++.

One of the minor defects found during the informal code review (KD_008), but not found during the field trial, deals with an unhandled raised exception. This type of problem was supposed to be detected during this step by browsing the Logiscope control flow diagrams. Since no exception handling problems were detected by this step, we conclude that browsing with Logiscope may not be an effective technique for novice Logiscope users.

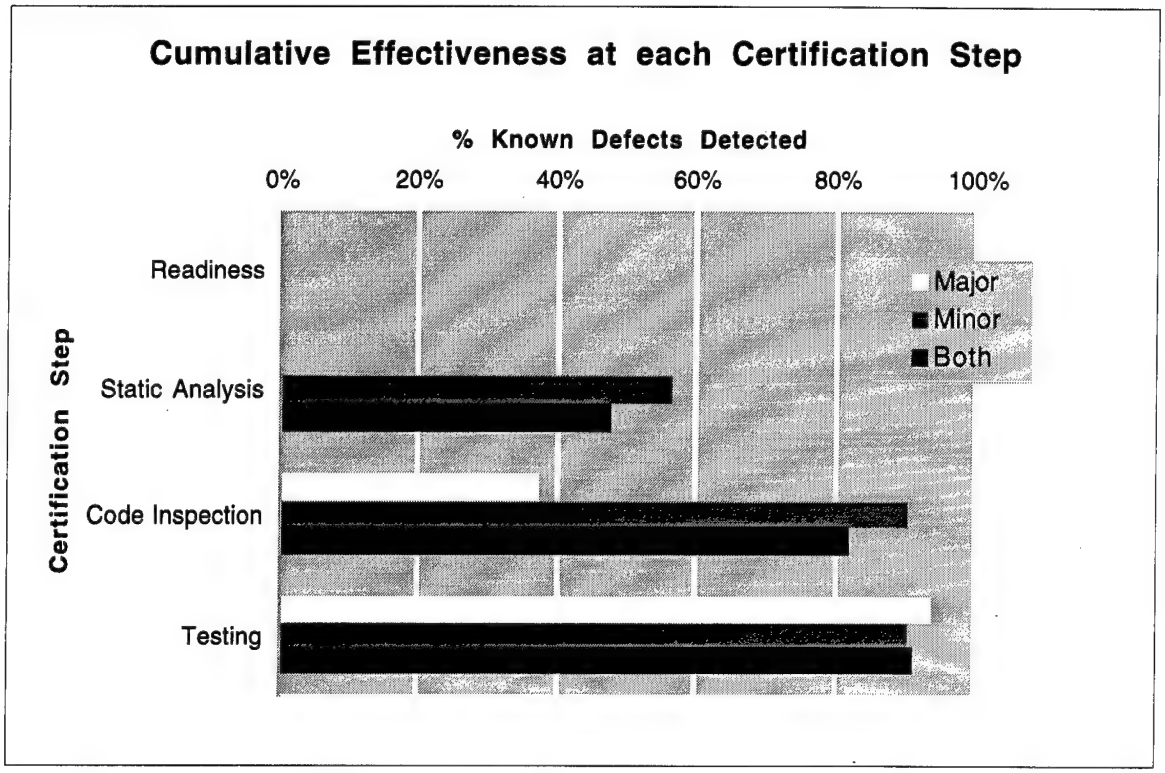


Figure 9-18. Cumulative effectiveness of certification steps

Code Inspection step. As Figure 9-18 shows, this step found about one third of the major errors. This was disappointing, and likely explanations are as follows:

- highly effective inspections reported in the literature are multi-person techniques and the certification process uses a single inspector technique
- the checklist approach focuses too much attention on the checklist at the expense of a deeper understanding of the code
- the inspection technique may be weak at finding logic defects

Using the initial version of the checklist, defects that had been found by the Static Analysis step were found and reported again in the Code Inspection step. The checklist in Section 2 of Volume 5 - Certification Field Trial has been edited to remove duplications, because automated static analysis is a much more cost effective way to find a defect.

Two of the minor defects found during the informal code review but not found in this step relate to variables that are declared but never referenced. Even though this is an inspection checklist item (D.01.C) that resulted in two other defect reports, it did not catch all occurrences. This is a perfect example of the type of analysis that could better be done with an automated static analysis tool.

Testing step. About two-thirds of the major defects were found in the testing step, as can be seen by subtracting the effectiveness of the code inspection step from that of the testing step in Figure 9-18. All defects found during the testing step were, by definition, considered major defects. As discussed previously under the Asset's Defect Profile paragraphs, we were unable to categorize most of the defects found during testing.

By removing the seeded defects one at a time, and reapplying the test cases, the Certification Analyst was able to attribute two of the nine defect reports resulting from testing to seeded defects. The remaining test failures may or may not be attributed to defects reported in other steps, or by more than one defect in combination. Without debugging these test failures, it is impossible to attribute them to defective lines of code. Therefore, the unattributed testing defect reports are counted as newly discovered natural defects.

Additional details about the certification field trial can be found in the supporting document titled CRC Volume 5 - Certification Field Trial.

9.6 Summary of the Code Defect Model

In the study of code defects, an empirically-based source code defect model was developed. The model is a predictive model of latent defects in software components. It describes the types of defects that are to be expected and can be detected and predicts the relative distribution and relative density of each type. The model also describes standard detection methods and for each method, predicts its effectiveness at finding each type of defect. Combined with estimates of the costs of applying the detection methods and removing defects, the model can be used in a cost/benefit analysis to determine the order in which methods should be applied in a certification process to maximize benefit, in terms of reduced risk of rework due to defects, and to minimize cost.

The source code defect model was developed using data extracted from the existing literature on software error studies. The studies from which data for the model was extracted fall into two general classes marked by whether or not all of the known defects were equally available to be detected by each method used in the study. In those cases where not all of the defects were available to all of the methods, it is possible that the percentage of defects detected by a particular method could have been higher if it had been applied against the full set of defects. In the other cases, it is not known what the overlap is between defects found by one method and those found by another since the same defect could have been found by more than one method. This is not necessarily important when comparing the effectiveness of one method to that of another method. It becomes important, however, when deciding how to select and sequence a range of methods based on how effective each is against certain types of defects *and* how frequently those types of defects occur.

The overlap uncertainty also complicates combining metrics for methods that are actually submethods or alternative techniques of a general class of methods into an aggregate metric for the general class. Thus, computing a metric for a general class of

methods like "testing" is more subject to error than computing a metric for a particular structural testing method like "data flow coverage". On the other hand, there is a more substantial base of data for the general classes of methods than there is for the particular types of techniques in those classes. This is due to a tendency for the studies to have focused either on one (or a few) methods or to have reported the combined results of all the activities of a whole development phase such as "testing". For this model, no metrics were aggregated from submethods for general classes of methods; the metrics given for general classes were computed directly from study data.

The source code defect model provides a starting point for selecting detection methods for a software certification process. Once the certification process is in place and being applied, data from that process should be collected and used to refine the model. Two areas of refinement are particularly recommended: (1) the defect classification schema should be expanded to include robustness defects, i.e., defects that result from reuse in a different context, and (2) data relevant to any overlap in the defects detected by the various methods should be collected, analyzed, and, if necessary, be used to adjust the detection method effectiveness ratings.

Additional details about the operational concept can be found in the supporting document titled CRC Volume 7 - Code Defect Model.

10 Lessons Learned

Several lessons were learned during the certification field trial. The lessons are categorized and described as follows:

- Installation and use of tools
- Evaluating certification techniques
- Certifier skills
- Effectiveness of techniques
- Modifications to the process guidance

Installation and Use of Tools

Installation of the tools was more difficult than expected, mainly because the tools were received from different vendors. Each licensed tool used the Flex license manager, but required different versions. Therefore we recognize the necessity of providing tool installation support for any certification pilot sites.

The AdaCAST tool was not compatible with the Rational Apex Ada environment and therefore could not be used in the field trial. The static analyzer AdaWise tool contained four analyses, and only two of four worked without error. The third analysis generated ASIS error messages and the fourth did not execute at all.

Logiscope. The Logiscope documentation was contained in four separate manuals, none of which was a user's guide. It was difficult to learn how to use because there was no step-by-step guidance. Therefore we recommend vendor training for any certification pilot sites that will use Logiscope. We also added an Appendix B to Volume 5 - Certification Field Trials, titled "Using Logiscope" to the field trial procedures guide to supplement the vendor documentation.

Logiscope generated error messages during archival and displayed incorrect and inconsistent results with its GUI tool. The vendor's technical support staff was unable to diagnose the problem via telephone consultation, and did not seem knowledgeable about the Ada language. This made it very difficult to determine how to increase test coverage and was one of the reasons the testing step did not achieve the 100% DDP coverage goal. We recommend investigating other dynamic analysis tools.

Evaluating Certification Techniques

In designing this certification process, we found it difficult to compare the effectiveness of techniques at detecting defects in published studies. Much data is published, but the studies are not very comparable because of variations in the application of techniques, implementation languages, size of asset, seeded vs. natural defects, incomplete

information, etc. See the CRC Volume 7 - Code Defect Model for more detailed information about the synthesis of published studies.

There is a need for a certification benchmark or test bed of assets with well documented defects against which techniques may be applied and their effectiveness established. Ideally, the test bed would be a rich source of defects of all types, with a known defect profile. The ProGen asset with seeded defects can contribute to this test bed if the uncategorized defects found in the testing step are debugged.

We found seeding defects to be more difficult than was originally anticipated, and were concerned that seeded defects might not typify natural defects, or might be trivial and thus more easily detected than natural defects.

Certifier Skills

The suite of certification techniques that comprise the default certification process includes two techniques whose effectiveness is highly dependent upon the training and experience of the certification engineer applying the technique: code inspection and testing. These techniques are also less automated and require more human involvement than the readiness and static analysis steps. This implies that the results may not be repeatable when comparing different certification engineers. To reduce the variability among different engineers, and to maximize the effectiveness of the techniques, training is essential.

The default process steps are intentionally ordered in terms of increasing skill level as well as increasing investment of effort, so that, for example, a failure in an early step could save wasted effort in later steps. In general, we would like the automated static analysis tools to detect as much as possible, and we view enhancements in static analysis capabilities as a valuable contribution to certification.

Effectiveness of Techniques

The combined effectiveness of all of the steps in the certification process is impressive because each step tends to find different types of defects. We had originally considered the following certification level scheme:

Certification Level	Certification Step
0	Readiness
1	Static Analysis
2	Code Inspection
3	Testing

As Figure 9-19 shows, for example, that all of the major defects would have been missed if we had only done a Level 1 certification. It also indicates that we would not have

wanted to jump into testing without having performed the preceding 3 steps. We now believe that a single-technique-per-level certification policy, which is typical of many reuse repositories, may not make sense. Instead, we believe that the techniques should be applied in combination.

This idea has affected our thinking about certification levels in the Certification Framework. The final version of the Certification Framework [see CRC Volume 2 - Certification Framework] proposes a two-dimensional view of certification levels: scope and rigor. Increased rigor of a testing technique, for example, would correspond to more stringent coverage criteria.

Modifications to the Process Guidance

General. We have specified both physical and logical lines of code on the asset size. We have clarified the instructions on how to report defects, so that defects can be counted more consistently, as follows:

- no more than one package per defect report
- all occurrences of the same defect, such as a style guideline violation or inspection checklist item, are recorded on the same defect report

Instructions in the procedures guidance of Section 2 of Volume 5 - Certification Field Trial were modified as described below.

Readiness step. Added instructions to check for superfluous files.

Static Analysis step. Removed reference to the AdaWise analysis that is currently not working.

Code Inspection step. Removed 10 checklist items that are automatically checked by AdaQuest in the Static Analysis step. Modified the wording of a few questions to be more Ada-specific.

Testing step. Added an appendix with instructions in using Logiscope. Removed references to the AdaCAST tool. Relaxed the test coverage goal to 90% DDP coverage and described potential exceptions (see subsection 2.6 of Volume 5 - Certification Field Trial). A coverage goal of 100% branch coverage is appropriate for unit testing, but may be impractical when testing a larger aggregate of software.

11 Conclusions

Whereas separate conclusions in each of the areas were discussed in the previous section, overall conclusions for the project are discussed in this section. These conclusions are an integration of the overall findings of the total effort.

Overall, much has been accomplished under the CRC project:

- An assessment of the state of the practice for reuse and certification and their supporting technologies.
- A Certification Framework that is adaptable to a wide variety of domains, business strategies and asset types.
- A Cost/Benefit Plan that uses rework to asses risk and show the economic value of certification in a reuse program.
- A certification cost model that optimizes certification benefits with respect to costs, tailored to an organization's requirements.
- A certification algorithm that defines the processes and tasks to isolate and analyze defects by type and severity.
- An evaluation of static analysis and testing techniques that can be used to create a certification environment that is site-specific.
- Results from initial certification field trial and detailed procedures and guidelines to perform succeeding certification field trials at different sites.
- Selected team members participated in the Reuse Library Interoperability Group (RIG) to develop an IEEE Standard for a method to specify certification policies.

These accomplishments of the CRC project have greatly advanced certification technology. For example, the CF provides a structure of elements to consider in a certification environment. The Cost/Benefit Plan provides a systematic approach for evaluating the cost and benefits of applying certification technology within a reuse program. Our approach maximizes rework avoidance with respect to a technique's defect detection effectiveness, investment cost, and incremental cost. The Operational Concept provides a user's perspective of a certification environment. The Certification Field Trail document provides a process to apply the CF, procedures, and collection forms. The field trial provided results and lessons learned.

In light of the CRC accomplishments, we performed assessments against each of the success measures for R&D projects and technology transition previously identified. For both these sets of measures, our assessment of our project accomplishments is very positive. As illustrated in Table 11-1, CRC has contributed to each of the R&D areas of

innovation, experimentation and validation. Likewise, CRC has contributed to each of the areas of technology transfer of awareness, communication and application as illustrated in Table 11-2. Examples of the activities that were accomplished in the area of awareness of CRC technology are listed in Table 11-3.

Table 11-1. Assessment of CRC measures of R&D success

R&D Measures	Certification Algorithms	Evaluation of Testing and Static Analysis Techniques	RIG Activities
Validation		X	
Experimentation		X	X
Innovation	X	X	

Table 11-2. Assessment of CRC measures of technology transfer success

Technology Transfer Measures	AF CARDS	ASSET	Navy DSRS	ELSA	COSMIC	UL	AF DSRS - Gunter BLSM
Application						X	X
Communication	X	X	X	X	X	X	X
Awareness	X	X	X	X	X	X	X

Table 11-3. Examples of CRC technology awareness

Date	Activity
25 May 94	IEEE/SUNY College of Technology Dual Use Conference
23 Jun 94	DISA/CIM Certification Guidelines Workshop
07 Jul 94	DoD Joint Program Review (CARDS Facility)
21 Jul 94	Col. Garretson, Army Reuse Focal Point, Pentagon
8-11 Aug 94	3rd Annual Reuse Education & Training Workshop
17-19 Oct 94	USC Focused Workshop on Reuse (Affiliates of Dr. Boehm)
16 Nov 94	DISA/CIM Project Briefing
22 Nov 94	Lt. Col. Pait, Air Force Reuse Focal Point, Pentagon
23 May 95	IEEE/SUNY College of Technology Dual Use Conference
13-14 June 95	ECS Architecture Technical Interchange Meeting (TIM)
30 Oct- 2 Nov 95	Applications of Software Measurement
14-15 Nov 95	U. S. Army CECOM Technical Interchange Meeting (TIM)
5-8 May 96	Institute of Operations Research and Management Science, Analysis to Support Public Sector Decision Making

12 Implications for Future Research

All of these conclusions, achievements and their lessons learned, however, should be viewed within the context of the phases and milestones of technology maturation as illustrated in Figure 12-1 [RED93]. Redwine investigated the growth and propagation of many software engineering technologies to characterize the conditions that facilitate their transfer to industry. His characterization can be applied to certification technologies and has implications for future research.

For example, during the Basic Research Phase, ideas and concepts are investigated that later prove fundamental, and there is a general recognition of problem and discussion of its scope/nature. In the Concepts Formulation Phase, informal ideas circulate and there is a convergence on a compatible set of ideas with a general publication of solutions to parts of the problem. The Development and Extension Phase has trial, preliminary use of the technology, clarification of the underlying ideas and an extension of the general approach to a broader solution. The Enhancement and Exploration (Internal) Phase brings a major extension of the general approach to other problem domains, the use of the technology to solve real problems, stabilization and porting of the technology, development of training materials and derivations of results indicting value.

The Enhancement and Exploration (External) Phase has the same activities as in Internal above, but the activities are carried out by a broader group, including people outside the development group. The Popularization (40%) Phase has the appearance of production-quality, supported versions, commercialization and marketing of the technology and propagation of the technology throughout community of users. And finally, the Popularization (70%) Phase has the same activities as in Popularization (40%) Phase, only with a larger following.

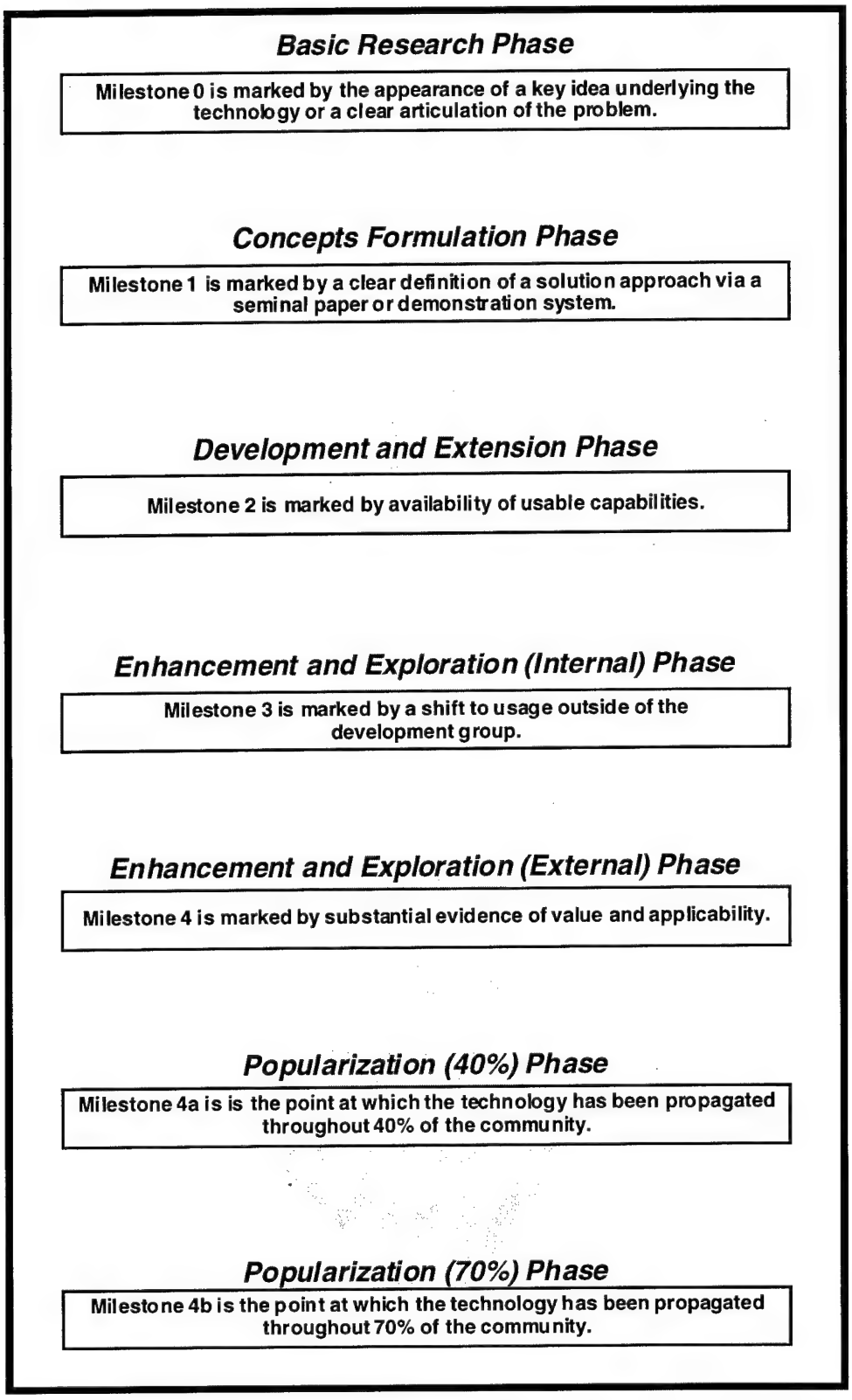


Figure 12-1. Phases and milestones for technology maturity [RED93]

Milestones within these stages can be marked by the occurrence of the following events:

1. Basic research - The recognition of a problem, an assessment of its scope, and the investigation of ideas and concept that may lead to a solution.

2. Concepts formation - The convergence toward a compatible set of ideas and the publication of solutions to parts of the problem.

3. Development and extension - The clarification and extension of the ideas, and the trial and preliminary use of the technology.

4. Enhancement and exploration - The portage, development of training materials and use of the technology to solve real problems.

5. Popularization - The appearance of production-quality, supported versions, and commercialization and marketing.

Redwine also found that the average timeframe for a technology to mature from Milestone 0 to Milestone 4a is approximately 15-20 years. Widespread use can take another decade.

We can apply Redwine's maturity profile to certification technology. We are of the opinion that certifying reusable software components was clearly in the Basic Research Phase prior to Milestone 0 at the initiation of the CRC project, even though some supporting reuse technologies were more mature (e.g., domain analysis, asset production, asset selection, and reuse libraries). We believe the products of CRC have helped advance certification technology into the phases of Concept Formulation and Development and Extension.

Within this roadmap to technology maturation, a plan for certification is feasible. Moving toward the Popularization Phase and beyond is achievable, but will require considerable time and effort. Therefore, those planning for reuse and certification must be sensitive to this proposed profile of technology maturity and the time required to achieve each milestone and transition through all phases.

Our recommendations for future certification programs are to extend our existing research by investigating a number of possible areas. For example, exploration of certification techniques for the remaining quality concerns is a natural progression of our initial work, as shown in Figure 12-2. Applying the CF to other quality concerns (such as robustness) and other asset types (such as architectures) appears promising. A study of the implication of business models, domains, asset types and quality factors on certification methods could further enhance our CF. An envisioned next step for future funded projects is to develop a plan for applying and validating the CF using different attributes for the elements of the reuse context.

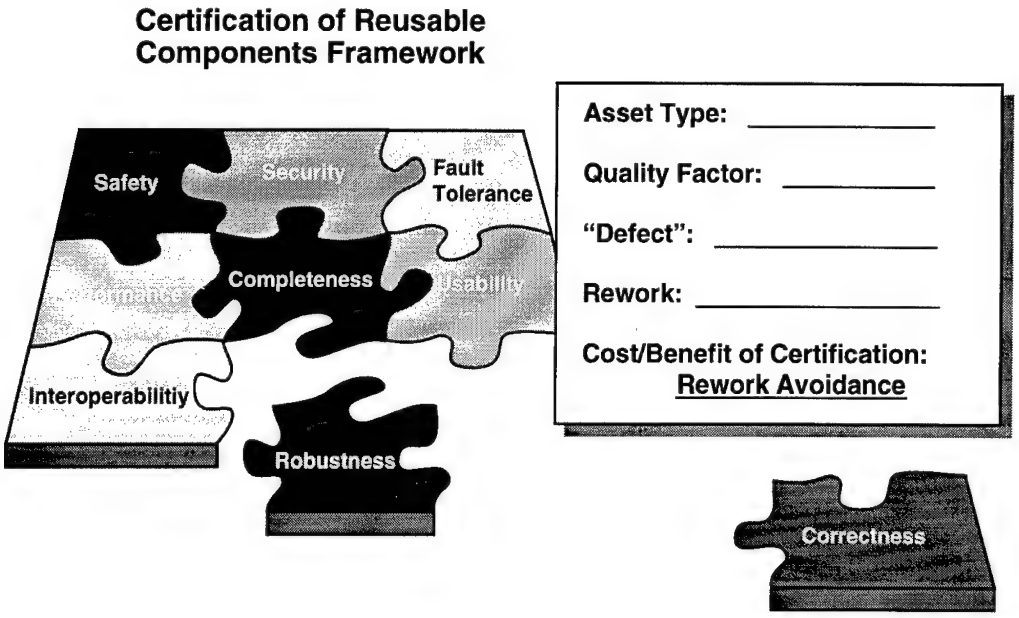


Figure 12-2. Extension of the CF to other quality concerns

Another area of future research is certification of fault tolerant systems, the methods and tools available, and their effectiveness with respect to required rigor levels. Valuable research could be conducted by investigating how certification methods can lead to standards for the development of reusable assets. Also, programmatic research could be performed by applying certification to a specific program. Likewise, additional pilot studies in varying domains could advance the maturation of certification technology. These possible extensions of our existing CF research and development could potentially influence not only DoD practices, but also national and international commercial practices through standardization of methods, tools and techniques.

References

[ANS94] ANSI/ASQC Q9000-1-1994, American National Standard, Quality Management and Quality Assurance Standards - Guidelines for Selection and Use, American Society for Quality Control, Milwaukee, Wisconsin, August 1, 1994.

[ARM95] U.S. Army Space and Strategic Defense Command Software Engineering Division, "Component Evaluation Procedure (Phase II) Technical Report, January 31, 1995.

[BAN93] Banker, Rajiv D., Robert J. Kauffman, Dani Zweig, "Repository Evaluation of Software Reuse," *IEEE Transactions on Software Engineering*, Vol. 19, No. 4, April 1993.

[BOE93b] Boeing Company, Defense & Space Group, "STARS Conceptual Framework for Reuse Processes (CFRP)," Volume I: Definition, Version 3.0, STARS-VC-A018/001/00, Seattle, WA, October 25, 1993.

[BOW85] Bowen, T.P., et. al. "Specification of Software Quality Attributes," Technical Report, RADC-TR-85-37, Rome Laboratory, February 1985.

[BUN94] Bundy, G.N. ., W.W. Agresti and W.R. Stewart, "Software Tool Support for Reuse Certification," The MITRE Corporation MTR94W0000109, September 1994.

[CAR94] Card, David N. and Edward Comer, "Why Do So Many Reuse Programs Fail?" *IEEE Software*, September 1994, p. 114-115.

[CHU93] Chubin, Sherrie and David Eichmann, David Card, Duane Hybertson, "Software Reuse Program, Software Metrics Plan," Defense Information Systems Agency, Joint Interoperability Engineering Organization, Center for Information Management, DISA/JIEO/CIM, Version 4.1, August 4, 1993.

[COM96] Comer, Edward R., P1420.1A/D6, Guide for Information Technology - Software Reuse - Asset Certification Framework, Technical Committee 4: Asset Evaluation and Certification of the Reuse Library Interoperability Group (RIG) January 1996.

[DIS94] Defense Information Systems Agency Center for Software, "Methods of Certifying Non-Code Reusable Assets," Detailed Report DCA 100-93-D-0066, December 1994.

[DIS95] Defense Information Systems Agency Center for Software, DoD Software Reuse Initiative, "Software Reuse Business Model (SRBM)," Technical Report, January 31, 1995.

[DOD94a] DoD Software Reuse Initiative (SRI), Technology Roadmap, Version 2.0, Volume 1: Technology Assessment, October 4, 1994.

[DUN92] Dunn, Michael F. , John C. Knight, "Certification of Reusable Software Parts," Department of Computer Science, University of Virginia, Charlottesville, VA, and the Software Productivity Consortium (SPC), INF-92-001, August 31, 1992.

[FRA95] Frakes, William B., and Christopher J. Fox, "Sixteen Questions About Software Reuse," *Communications of the ACM*, June 1995, Vol. 38, No 6, pp. 75-87.

[GAO93] General Accounting Office. "Software Reuse: Major Issues Need to Be Resolved Before Benefits Can Be Achieved." GAO/IMTEC-93-16, January 1993.

[HES90] Hess, James A., William E. Novak, Patrick C. Carroll, Sholom G. Cohen, Robert R. Hollbaugh, Kyo C. Kang, A. Spencer Peterson, "A Domain Analysis Bibliography," Carnegie-Mellon University and the Software Engineering Institute, Special Report, CMU/SEI-90-SR-3, 1990.

[HOO91] Hooper, James W. and Rowena O. Chester, *"Software Reuse, Guidelines and Methods,"* Plenum Press, New York, NY, 1991.

[INT91] International Standard ISO/IEC 9126, Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for the Use, International Organization for Standardization, International Electrotechnical Commission, 1991.

[JON95] Jones, Capers, "Return on Investment in Software Measurement," Proceedings of the 6th International Conference on Applications of Software Measurement (ASM), October 30-November 2, 1995, Orlando, FL.

[MCI69] McIlroy, M.D., "Mass -Produced Software Components," Software Engineering Concept and Techniques: 1968 NATO Conference on Software Engineering, J.M. Buxton, P. Naur, and B. Randell, eds., Petrocelli/Charter, New York, 1969, pp. 88-98.

[MIL95] Miller, L.A., J.E. Hayes, and S. Mirsky, *Guidelines for Verification and Validation of Expert System Software and Conventional Software*, NUREG/CR-6316, U.S. Nuclear Regulatory Commission, March 1995.

[MIT95] MITRE Corporation, "High Integrity Software for Nuclear Power Plants: Candidate Guidelines, Technical Basis and Research Needs," NUREG/CR-6263, MTR 94W0000114, Volumes 1 & 2, June 1995.

[NAT95] National Software Data & Information Repository (NSDIR), "Metrics Collection and Submission Guide, " Volume I - General Instructions, Volume II - Repository Information Request, Volume III - Recurring Data Form, Deputy Assistant Secretary for Communications, Computers, and Logistics, Office of the Assistant Secretary of the Air Force for Acquisition, Version 2.0, 16 June 1995.

[NRC93] Nuclear Regulatory Commission (NRC), "Operating Reactors Digital Retrofits, Digital System Review Procedures," Version 1, Digital Systems Reliability and Nuclear Safety Workshop, 1993.

[POO92] Poore, J.H., Theresa Pepin, Murali Sitaraman, Frances L. Van Scoy, "Criteria & Implementation Procedures for Evaluation of Reusable Software Engineering Assets," Software Technology For Adaptable, Reliable Systems (STARS) Program, Task/Subtask IT00.19, CDRL Sequence 04014-002B, July 16, 1992.

[PRI93] Prieto-Diaz, Ruben, "Status Report: Software Reusability," *IEEE Software*, May 1993, pp. 61-66.

[RAT94] Rathbun, Robert W. "Software Reuse Metrics," Proceedings of the Software Technology Conference (STC), Volume 1, Salt Lake City, UT, 1994.

[RED93] Redwine, S.T. and M.M. Eward, "Software Engineering Technology Transfer Practices," International Perspectives in Software Engineering, January 1993, pp. 18-22.

[SAM95] Samadzadeh, Mansur and Mansour Zand edited the proceedings Confessions of a Used-Program Salesman: Lessons Learned, *Proceedings of the Symposium on Software Reusability , SSR'95"* Seattle, WA, April 28-30, 1995.

[SPS94] Software Productivity Solutions, Inc. "Component Certification: State-of-the-Art Technology Report," Subcontract No. P48124 under Prime Contract No. F30602-92-C-0158, U. S. Air Force Rome Laboratory, RL/C3CB, Rome, NY, November 1, 1994.

[TRA87] Tracz, Will, "Reusability Comes of Age, *IEEE Software*, July 1987, pp. 6-8.

Other Documents Used

[ADE85] Adelson, Beth and Elliot Soloway, "The Role of Domain Experience in Software Design, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November 1985, pp. 1351-1360.

[ARA89] Arango, G. "Domain Analysis - From Art Form to Engineering Discipline," *Proceedings of the 5th International Workshop of Software Specifications and Design*, p. 152-159, 1989.

[ARA95] Arango, Guillermo, "Software reusability and the Internet, " *Proceedings of the Symposium on Software Reusability , SSR'95,* edited by Mansur Samadzadeh and Mansour Zand, Seattle, WA, April 28-30, 1995.

[BAI88] Bailin, Sidney, "Semi-Automatic Development of Payload Operations Control Center Software," NASA Goddard Space Flight Center, Computer Technology Associates, Laurel, MD, October 1988.

[BAI89] Bailin, Sidney, "Generic POCC Architectures," NASA Goddard Space Flight Center, Computer Technology Associates, Laurel, MD, April 1989.

[BAR91] Barnes, B. and T. Bollinger, "Making Reuse Cost-Effective," *IEEE Software,* 8(1), p. 13-24.

[BAT88] Batory, Don S., J.R. Barnett, J. Roy, B.C. Twichell and Jorge F. Garza, "Construction of File Management Systems for Software Components," Technical Report TR-88-36, University of Texas, Austin, TX, October 1988.

[BER95] Bergstrom, Deane, "Certification of Reusable Software Components," Briefing chart in response to Project Overview, December 12, 1995, Rome Laboratory, NY.

[BIE95] Bieman, James M. and Santhi Karunanithi, "Measurement of Language-Support Reuse in Object-Oriented and Object-Based Software," *Journal of Systems Software*, 1995: 30: pp. 217-293.

[BIG87] Biggerstaff, Ted and Charles Richter, "Reusability Framework Assessment and Directions," *IEEE Software*, March 1987.

[BIG88] Biggerstaff, Ted. J. , "The Nature of Semi-Formal Information in Domain Models," Technical Report STP-289-88, Microelectronics and Computer Technology Corporation, Austin, TX, September 1988.

[BIG89] Biggerstaff and Alan J. Perlis, *"Software Reusability, Volume I, Concepts and Models* and *Volume II, Applications and Experience,"* ACM Press New York, NY, 1989.

[BOE91] Boeing Company, Defense & Space Group, US40 "STARS Reuse Concept of Operations," Volume I, Version 0.5, Draft, Informal Technical Data, STARS-SC-03725/001/00, Seattle, WA, August 27, 1991.

[BOE93a] The Boeing Company, "Reuse Strategy Model: Planning Aid for Reuse-based Projects," Software Technology For Adaptable, Reliable Systems (STARS) office, 9-5526, F19628-88-D-0028, Task U03, CDRL 5159, July 31, 1993.

[BOR84] Borgida, Alexander, John Mylopoulos and Harry K.T. Wong, "Generalization/Specialization as a Basis for Software Specifications," *On Conceptual Modeling*, pp. 87-117, Springer-Verlag, New York, NY 1984.

[BRO92] Brown, Linda, "DOD Software Reuse Initiative, Vision and Strategy, OASD(C3I)/DDI, July 15, 1992.

[BRU88] Bruns, Glen and Colin Potts, "Domain Modeling Approaches to Software Development, " Technical Report, STP-186-88, Microelectronics and Computer Technology Corporation, Austin, TX, June 1989.

[CAC95] CACI, Inc. - Federal, "Systems Engineering and Technical Support for DISA/Center for Software," Procedures for Qualification and Engineering of Reusable Assets (Final), U.S. Department of Defense, Defense Information System Agency, Arlington, VA, 1995.

[CAL91] Caldiera, Gianluigi and Victor R. Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer*, February 1991, pp. 61-70.

[CAR87] Carle, Rick, "Reusable Software Components for Missile Applications," *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse*, Syracuse University and University of Maryland, Blue Mountain Lake, NY, July 1987.

[CHA91] Chidamber, Shyam R. and Chris F. Kemerer, "Towards a Metrics Suite for Object-Oriented Design," OOPSLA '91, pp. 197-211.

[COH89] Cohen, Joel, "Software Reuse for Information Management Systems," Position Paper of the Reuse in Practice Workshop, Software Engineering Institute (SEI), Pittsburgh, PA, July 1989.

[COH92] Cohen, Sholom, Jay L. Stanley, Jr. A. Spencer Peterson, Robert W. Krut, "Application of a Feature-Oriented Domain Analysis to the Army Movement Control Domain," Technical Report, CMU/SE 1-91-TR-28 ESD-TR-91-28, June 1992.

[COM95] Comer, Edward R., P1420.1/D5, Guide for Information Technology - Software Reuse - Asset Certification Framework, Technical Committee 4: Asset Evaluation and Certification of the Reuse Library Interoperability Group (RIG) September 1995.

[DOD94b] DoD Software Reuse Initiative (SRI), Technology Roadmap, Version 2.0, Volume 2: Implementation Plan, October 4, 1994.

[DOD95a] Department of Defense, "Software Reuse Symposium," March 23, 1995, Huntsville, Alabama.

[DOD95b] Department of Defense, "Domain Scoping Framework", Version 3.1, Volume 2: Technical Description, 29 September 1995.

[FAC94] Facemire, J. Jeff, Aleisa Petracia and Stephen Riesbech, "Software Architecture Seminar Report," Software Technology for Adaptable Reliable Systems (STARS), Central Archive for Reusable Defense Software (CARDS), Informal Technical Report, Contract No F19628-93-C-0130, January 29, 1994.

[FIS87] Fischer, Gerhard, "Cognitive View of Reuse and Redesign," *IEEE Software*, July 1987.

[FOW95] Fowler, Glenn S., David G. Korn and Kiem-Phong Vo, "Principles for Writing Reusable Libraries," *Proceedings of the Symposium on Software Reusability , SSR'95,* " Seattle, WA, April 28-30, 1995.

[FRA92] Frakes, William, Ruben Prieto-Diaz and Edward Comer, "Ada Software Reuse and Domain Analysis Seminar, presented at the Clarion Plaza Hotel, Orlando, FL, November 16, 1992.

[GAL95] Gall, Harald, Mehdi Jazayeri and Rene Blosch, "Research Directions in Software Reuse: Where to go from here?", *Proceedings of the Symposium on Software Reusability (SSR '95)* , edited by Mansur Samadzadeh and Mansour Zand, Seattle, WA, April 28-30, 1995.

[GIL89] Gilroy, Kathleen, Edward Comer, J. Kaye Grau, Patrick Merlet, "Impact of Domain Analysis on Reuse Methods," Final Report CO4-087LD-0001-00, U. S. Army Communications - Electronics Command, Ft. Monmouth, NJ, November 1989.

[HUT88] Hutchinson, J.W. and P.G. Hindley, "A Preliminary Study of Large Scale Software Reuse," *Software Engineering Journal*, Vol. 3, No. 5, pp. 208-212. 1988.

[ISC88] Iscoe, Neil, "Domain-Specific Reuse: An Object Oriented and Knowledge-Based Approach" in Will J. Tracz, *Software Reuse: Emerging Technology*, pp. 299-308, IEEE Computer Society, Washington, D.C., 1988.

[JAC93] Jackelen, George and Larry McCutchan, PRISM Documentation Library, 1.0, Central Archive for Reusable Defense Software (CARDS)," Software Technology for Adaptable Reliable Systems (STARS), STARS-VC-B007/000/01, December 3, 1993.

[JAW90] Jaworski, Allan, Fred Hills, Thomas A. Durek, Stuart Faulk, John E. Gaffney, "A Domain Analysis Process," Interim Report 90001-N (Version 01.00.03), Software Productivity Consortium (SPC), Herndon, VA, January 1990.

[JOH88] Johnson, Ralph E. and Brian Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, June/July 1988, Vol. 1, No. 2, pp. 22-35.

[JON84] Jones, T.C. "Reusability in programming: A Survey of the State of the Art," *IEEE Transactions in Software Engineering*, pp. 488-494. September 1984.

[KAN89] Kang, Kyo C., "Features Analysis: An Approach to Domain Analysis," Position Papers of the Reuse in Practice Workshop, Software Engineering Institute (SEI), Pittsburgh, PA, July 1989.

[KAN90] Kang, Kyo C., and Sholom G. Cohen, James A. Hess, William E. Novak and A. Spencer Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical Report, CMU/SEI-90-TR-21, ESD-90-TR-222, November 1990.

[KAT94] Katz, Susan, and Christopher Dabrowski, Kathryn Miles, Margaret Law, NIST Special Publication 500-222, "Glossary of Software Reuse Terms," Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-0001, December 1994.

[LAT89] Latour, Larry, "Issues Involved in the Content and Organization of Software Component Information Bases, Interim Report," Technical Report for the U.S. Army CECOM, prepared by the University of Maine, Orono, ME, May 1989.

[LEE88] Lee, Kenneth J. et. al., "An OOD Paradigm for Flight Simulators," 2nd edition, Technical Report CMU/SEI-88-TR-30, Software Engineering Institute (SEI), Pittsburgh, PA, September 1988.

[LEN87] Lenz, Manfred, Hans Albrecht Schmid and Peter F. Wolf, "Software Reuse Through Building Blocks," *IEEE Software*, July 1987.

[LIM94] Lim, Wayne C., "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software*, September 1994.

[LUB87] Lubars, Mitchell D., "A Knowledge-Based Design Aid for the Construction of Software Systems," Ph.D. Thesis, University of Illinois at Urbana, Champaign, IL, 1987.

[LUB88] Lubars, Mitchell D., "Domain Analysis and Domain Engineering in IDeA," Technical Report STP-295-88, Microelectronics and Computer Technology Corporation, Austin, TX, September 1988.

[MAT84] Matsumoto, Yoshihiro, "Some Experiences in Promoting Reusable Software Presentation in Higher Abstract Levels," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 502-513.

[MCC85] McCain, Ron, "A Software Development Methodology for Reusable Components," *Proceedings of the Software Technology for Adaptable Reliable Systems (STARS) Workshop*, pp. 361-384, Naval Research Laboratory, Washington, D.C., April 1985.

[MCN86] McNicholl, Daniel G., et. al. "Common Ada Missile Packages (CAMP) Vol. 1, Overview and Commonality Study Results," Technical Report AFATL-TR-85-93, McDonnell Douglas Astronautics Company, St. Louis, MO, May 1986.

[MER93] Merritt, Steven, "Framework for Certification of Reusable Software Components," DISA/CIM Software Reuse Program, February 26, 1993.

[MEY87] Meyer, Bertrand, "Reusability: The Case for Object-Oriented Design," *IEEE Software*, March 1987. pp. 50-63.

[MOR89] Moore, John A. and Sidney C. Bailin, "Domain Analysis: Framework for Reuse," Technical Report, Computer Technology Associates, Rockville, MD, October 1989.

[MOS95] Moseman, Lloyd K., "Software Development: Quo Vadis?" *Crosstalk*, November/December 1995, Volume 8, Number 11, p. 2-3.

[MYE88] Myers, Brad, "A Taxonomy of Window Manager User Interfaces," *IEEE Transactions on Computer Graphics and Applications*, Vol. 8, No. 5, pp. 65-84. September 1988.

[NEI80] Neighbors, James, "Software Construction Using Components," Ph.D. Thesis, University of California at Irvine, CA, 1980.

[NEI83] Neighbors, James, "The DRACO Approach to Constructing Software from Reusable Components," *Proceedings of the Workshop on Reusability in Programming*, ITT Programming, Stratford, CT, September 1983, pp. 167-178.

[PAR76] Parnas, David, "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, March 1976, pp. 1-9.

[PAR79] Parnas, David, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, March 1979, pp. 128-138.

[PAR85] Parnas, David, Paul C. Clements and David Wise, "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, March 1985, pp. 259-266.

[PAY88] Payton, Teri F., "Reusability Library Framework," Presentation at STARS Foundations Workshop, Unisys Defense Systems, Paoli, PA, April 1988.

[PER89] Perry, James M. and Mary Shaw, "The Role of Domain Independence in Promoting Software Reuse: Architectural Analysis of Systems," Position Paper of the Reuse in Practice Workshop, Software Engineering Institute (SEI), Pittsburgh, PA, July 1989.

[PET93] Petracca, Aleisa, Les Hayhurst and George Jachelen, "Portable, Reusable, Integrated Software Modules (PRISM) Documentation Library Model, Document Release 1.0, Central Archive for Reusable Defense Software (CARDS), Software Technology for Adaptable Reliable Systems (STARS), STARS-VC-B015/000/00, December 3, 1993.

[POU95] Poulin, Jeffrey S. and Keith J. Werkman, "Melding Structure Abstracts and the World Wide Web for Retrieval of Reusable Components," *Proceedings of the Symposium on Software Reusability (SSR'95)*, edited by Mansur Samadzadeh and Mansour Zand, Seattle, WA, April 28-30, 1995.

[PRI87a] Prieto-Diaz, Ruben, "Domain Analysis for Reusability," *Proceedings of the COMSAC 87: The Eleventh Annual International Computer Software and Application Conference*, pp. 23-29. IEEE Computer Society, Washington, D.C., October 1987.

[PRI87b] Prieto-Diaz, Ruben, "Faceted Classification and Reuse Across Domains," *Proceedings of the Workshop on Software Reuse*, Rocky Mountain Institute of Software Engineering, Boulder, CO, October, 1987.

[PRI91] Prieto-Diaz, Ruben and Guillermo Arango, *Domain Analysis and Software System Modeling*, IEEE Computer Society Press, Los Alamitos, CA, 1991, ISBN 0-8186-8996-X, p. 63-69.

[PYS92] Pyster, Art, "Reuse Adoption Guidebook," Software Productivity Consortium, SPC-92051-CMC, Version 01.00.03, November 1992.

[RTI93] Research Triangle Institute, "Certification of Reusable Software Components," U.S. Air Force Rome Laboratory, Contract No. F30602-92-C-0158, March 1993.

[SIM87] Simos, Mark A., "The Domain-Oriented Software Life Cycle" Towards an Extended Process Model for Reusability," *Proceedings of the Workshop of Software Reuse*, Rocky Mountain Institute of Software Engineering, Boulder, CO, October 1987.

[SIM95a] Simos, Mark and Dick Creps, "ODM (Organization Domain Modeling) Guidebook Published," *STARS Newsletter* (Software Technology for Adaptable, Reliable Systems), March 1995, Issue 12, p. 11.

[SIM95b] Simos, Mark and Dick Creps, Carol Klingler, Larry Levine, "Organization Domain Modeling (ODM) Guidebook, Version 1.0, STARS-VC-A023/911/00 Informal Technical Report, Contract No F19628-93-C-0130, March 17, 1995.

[SOL89] Solderitsch, James J. , Kurt C. Wallnaw and John A. Thalhamer, "Constructing Domain-Specific Ada Reuse Libraries," *Proceedings of the Seventh Annual Conference on Ada Technology*, U.S. Army CECOM, Ft. Monmouth, N.J., March 1989, pp. 419-433.

[SPC91a] Software Productivity Consortium (SPC), "Domain Analysis Workshop Presentations," SPC-91186-MC Version 01.00.00, September 26-27, 1991, Herndon, VA.

[SPC91b] Software Productivity Consortium, "Synthesis Workshop," September 23-25, 1991. Herndon, Virginia.

[SPC92a] Software Productivity Consortium, "Synthesis Guidebook," Volume 1. SPC-92111-CMC, Version 01.00.00 October 1992.

[SPC92b] Software Productivity Consortium, "Synthesis Guidebook, Volume 2: Case Studies," SPC-92111-CMC, Version 01.00.00 October 1992.

[SPS95] Software Productivity Solutions, Inc., *Task Area: Software Quality Framework (SQF)*, Interim Technical Report for the U.S. Air Force Rome Laboratory, Contract No., F30602-92-C-0158, October 1995.

[TRA93] Trail, Glen and George Jachelen, "Portable, Reusable, Integrated Software Modules (PRISM) Documentation of Library User's Guide," Release 1.0, Central Archive for Reusable Defense Software (CARDS), Information Technical Report from the Software Technology for Adaptable Reliable Systems (STARS) Program, STARS-VC-B006/001/101, December 3, 1993.

[VCO93] Virginia Center of Excellence for Software Reuse and Technology Transfer (VCOE) "Reuse Adoption Guidebook," Software Productivity Consortium, SPC-92051-CMC, Version 02.00.05, November 1993.

[WAR88] Ward, Paul T. and Lloyd G. Williams, "Using the Structured Techniques to Support Software Reuse," *Proceedings of the Structured Development Forum,* San Francisco, CA, August 1989, pp. 211-222.

[WEI88] Weiss, David, "Reuse and Prototyping: A Methodology," Technical Report SPC TR-88-022, Software Productivity Consortium (SPC), Reston, VA, March 1988.

Appendix A - Annotated Bibliography of Business Strategies

The information gleaned from this literature survey of business strategies was used to determine the operational context of the Reuse Context for Asset Quality Certification and to assess the impact of this previous research on the development of the Certification Framework.

The annotations in this appendix summarize the essence of each of the referenced publications. Summaries vary in length; those that are longer provide additional details because the reference appeared to be a flagship among others. The shorter annotations were still included to serve as a pointer to the complete reference if more details are of interest.

This annotated bibliography this area is not exhaustive, but gives a flavor of the previous research that has been accomplished. Some of these references were used in other appendices.

[BAN93] Banker, Rajiv D., Robert J. Kauffman, Dani Zweig, "Repository Evaluation of Software Reuse," *IEEE Transactions on Software Engineering*, Vol. 19, No. 4, April 1993.

Banker indicated that reuse, by it nature, is an activity that spans multiple projects and application systems within and across enterprises. To manage such reuse requires monitoring software at the level of the organization or enterprise rather than at the traditional focus of the individual software project control.

Banker found that organizational barriers and disincentives to reusing software were more serious than technical barriers to reuse. In general, he finds a lack of formal incentives to reuse objects. Software reuse is encouraged, but not mandated. Programmers are not rewarded for reuse. In fact, informal incentives exist for a programmer to prevent others from reusing their code. The creator is seen as the "owner" and becomes responsible for maintenance, even in applications for which it was not originally intended.

Programmer to programmer reuse is usually done by templating, as a hidden form of reuse which is not captured by traditional monitoring mechanisms. Templating achieves only some of the goals of software reuse; the coding effort and unit testing are reduced, but the adaptation costs are higher and subsequent life cycle savings, particularly in the maintenance phase, are not realized.

Banker maintains that the organization needs to maintain its software and related information in a repository software (i.e., design, history, interactions with other system elements, etc.). Rathbun believes that successful reuse relies upon effective repository cataloguing and searching as well as upon formal domain analysis.

By analyzing software at the repository level, one can cut across multiple projects to ask such questions as: "What kinds of objects are most likely to be reused and under what conditions?" He recommends focusing on development and process-oriented questions rather than a single, isolated product.

[BAR91] Barnes, B. and T. Bollinger, "Making Reuse Cost-Effective," *IEEE Software*, 8(1), p. 13-24.

Barnes writes that the scope of reuse can vary; reuse can be confined to a few specific methods and libraries of scavenged parts, or it can be broad to include the entire software development process and its artifacts. Requirements specifications, designs, code modules, documentation, test data, customized tools, and early work products are all candidates for reuse. Barnes believes that reuse should not be restricted to source code, since broad spectrum reuse has a greater potential to reduce costs. Modules that solve difficult or complex problems are good choices for reuse.

Barnes believes that effective reuse is one of the fundamental paradigms of development and needs to be better understood. Scavenging is the commonly

practiced scope of reuse, and is extremely inefficient. Lack of reuse planning duplicates re-engineering costs. Instead, effective reuse must be built into an organization's software development process. Barnes feels that industry needs consistent, broad-spectrum methodologies that integrate reuse analysis and development methods.

Barnes believes that the defining characteristic of good reuse is not the reuse of software per se, but the reuse of human problem solving. Human problem solving is the non-repetitive, non-trivial aspect of software development and maintenance that cannot be easily formalized or automated. It is a scarce and critical resource, it cannot be easily multiplied, multiplexed, accelerated or enhanced.

He believes that three techniques can be judiciously used to optimize human problem solving:

1. Planning
2. Automation
3. Reuse

Planning minimizes redundant and dead-end work while automation relies upon building tools to support manual tasks. Reuse can increase the effectiveness of human problem solving by recycling existing work in new contexts. Reuse should complement automation in tools, not compete with it.

Reuse has the same cost and risk characteristics as any financial investment; it can be viewed in the context of a consumer - producer model. The producer aspect of the model represents all the investments made to increase reusability. On the other hand, the consumer aspect of the model shows the cost benefits accrued as a result of the reuse investments. The consumer aspect of the model is concerned with the measure of dollars used and how earlier reuse investments have helped or hurt the final product. Likewise, Barnes believes that the producer and consumer model and its underlying processes can be applied to developing software reuse.

Reuse investment is cost-effective only when the total reuse investment R is less than the total cost benefits, B; that is $R < B$. Barnes proposes that if estimates of B are small, then R should be small. If estimates of B are large, then R should be large. The reuse presents a dilemma when organizations are faced with the risk associated with large investments in reuse without the guarantee of quantifiable large returns on their investment.

Reuse investments are most likely to pay off when they are applied to high-value work products. A rule of thumb is to build reusable parts if local expertise exists; purchase reusable parts if development requires expertise outside the organization. Moreover, coordinated reuse investments should be encouraged. Organizations that do not provide management incentives for reuse are likely to fail.

Barnes purports that is not a trivial concept; reuse has suffered from an image problem since it is usually viewed as a process of selecting from a salvage yard of software components. More appropriately, reuse can be a mechanism for preserving and guiding the use of expansive resources; that is, human creativity and ingenuity.

[BOE93a] The Boeing Company, "Reuse Strategy Model: Planning Aid for Reuse-based Projects," Software Technology For Adaptable, Reliable Systems (STARS) office, 9-5526, F19628-88-D-0028, Task U03, CDRL 5159, July 31, 1993.

The Boeing Company, under the STARS program, published a planning aid for projects beginning to institute reuse. This document describes a Reuse Strategy Model (RSM) that consists of a set of dimensions to characterize current reuse practices, a suggested process for performing the characterization, and a set of goals that are reasonable to adopt based on the current characterization. A prototype of the RSM was used by the STARS Demonstration teams and improvements were made resulting in this current version.

Historically, since the SEI's production of the Capability Maturity Model (CMM), there has been a growing interest in a similar model for the practice of reuse. The first broad-brush view of a reuse CMM was made public by the SPC as "Mount Reuse" in a presentation by K.V. Bourgeois titled "Technology Transfer of Mature Reuse Practice," found in the Proceedings of the Fifth Annual Workshop on Software Reuse, in October, 1992. Mount Reuse depicted five increasing stages of reuse maturity as ledges on a mountain side with "ad hoc" reuse at the bottom and "systematic" reuse at the top. Each level or ledge was annotated with characteristics of that stage.

The work at SPC was paralleled by Koltun and Hudson with their Harris Reuse Maturity Framework (HRMF) published in an article titled "A Reuse Maturity Model" in the Proceedings of the Fourth Annual Workshop on Software Reuse in November 1991. This model also had five stages labeled initial/chaotic, monitored, coordinated, planned and ingrained with ten dimensions across those stages.

The STARS' RSM evolved from work on the development of the Conceptual Framework for Reuse Processes (CFRP). The CFRP team, with members from representatives from Boeing, IBM, MITRE, Paramax, SEI and TRW, analyzed the Mount Reuse concept and concluded that a reuse maturity model would be needed to complement the CFRP to provide strategy planning guidelines.

In June 1992, a "Reuse Adoption" workshop was sponsored by SPC's DARPA contract for the Virginia Center of Excellence for Reuse (VCOE). A draft was presented similar to SEI's CMM concept of identifying key practice areas for each of five levels of capability maturity. Based on workshop feedback, the SPC rethought its approach and formulated a Reuse Capability Model (RCM) which is described in its Reuse Adoption Guidebook, annotated as [PYS92] and [VCO93] in this appendix.

During the development of SPC's RCM, the STARS program developed a prototype Reuse Strategy Model (RSM) to provide focused guidance to the STARS Demonstration teams in their reuse planning. The prototype was structured to support identifying project goals and metrics to be used in developing a reuse-based strategy that furthers achieves the STARS vision of reuse. The prototype was used on a trial basis and was improved and extended by adding a description of a process for applying it.

The resultant RSM document is designed for business and project planners in organizations who are transitioning to a domain-specific, reuse-based software development paradigm. The RSM helps planners set goals for achieving a state of practice compatible with the STARS vision of domain-specific reuse as articulated in the STARS Conceptual Framework for Reuse Processes.

The RSM identifies areas in which organizational objectives, policies, procedures, and process definitions can be applied to a project for a cost-effective reuse strategy. The RSM assesses the elements of reuse being practiced and suggests goals and metrics for monitoring progress against the goals.

The RSM is a matrix of five dimensions with thirty four indicators. Each dimension focuses on one aspect of reuse practice. The following five dimensions and its number of indicators are identified below:

1. Domain stability (5 indicators)
2. Organization readiness (9 indicators)
3. Experience with domain-specific knowledge (6 indicators)
4. Usage of technology for reuse processes (8 indicators)
5. Business climate and reuse management (6 indicators)

The process to apply the RSM begins with the domain of the project and characterizing the primary project goals with respect to the CFRP. The assessment is conducted using a Goal-Question-Metrics (GQM) paradigm and a goal is identified for each indicator. The goals are evaluated and prioritized relative to the project's context and constraints. The highest priority reuse goals are selected, with the progress metrics tailored and integrated into the project plans. Detailed descriptions of the process to apply the RSM are provided with suggested sample worksheets for every indicator.

[BRO92] Brown, Linda, "DOD Software Reuse Initiative, Vision and Strategy, OASD(C^3I)/DDI, July 15, 1992.

Linda Brown reports that DoD has evidence that software reuse principles, when integrated into acquisition practices and software engineering processes, provide a basis for dramatic improvement in the way software-intensive systems are developed and supported over their life cycle. She describes the vision and

strategy for a DoD initiative which is designed to make a reuse-based paradigm the preferred alternative for developing and supporting software. The strategy to realize this DoD vision is based upon systematic reuse; that is, opportunities are pre-defined and a process for capitalizing on those opportunities is planned and specified, not ad hoc. Brown believes that software reuse will eventually happen whether the DoD takes an active role or not. The challenge is to position the DoD to accelerate its use and to reap its benefits.

An infrastructure investment to encourage effective reuse includes advancing technologies that support reuse, incorporating reuse into existing management and current processes, and creating a generic set of components to use and reuse in new systems or in software maintenance. Domain analysis, domain models, and generic architectures are the primary focus of a successful reuse program. Near term cost savings will be offset by infrastructure investments. Other engineering disciplines have benefited by standard concepts, processes, and components allowing prior accomplishments to be leveraged and speed innovation for future systems. A similar strategy for reuse is proposed by the DoD.

A reuse-based software engineering process is based on four fundamental principles: domain-specific reuse, process-driven reuse, architecture-centric investment, and interconnected libraries. The Ada programming language provides a foundation upon which to base reuse efforts on a code level. However, Brown emphasizes that reuse is a process, not an end-product.

Brown describes the DoD reuse strategy as consisting of the following activities:

1. Establish domains.
2. Define reuse products (e.g., domain model, software architecture, product design, implementation components).
3. Establish criteria for deciding ownership.
4. Integrate reuse into the development and maintenance process.
5. Define the model for business decisions.
6. Define metrics to evaluate reuse success.
7. Define component guidelines for different reuse products.
8. Identify technology-based investment strategy (i.e., use of tools, knowledge representation, information systems security, etc.).
9. Conduct education and training.
10. Provide near term product and services (i.e., reuse maturity model).

The most important step in the DoD reuse strategy is the first step - to establish domains by defining boundaries. Domain analysis focuses on "areas of business" for initial domain decomposition.

Brown concludes that there is no singular approach to software reuse. Libraries facilitate, but do not enable reuse. This initial report has been recently updated in January 1995 as a working draft.

[CAR94] Card, David N. and Edward Comer, "Why Do So Many Reuse Programs Fail?" *IEEE Software*, September 1994, p. 114-115.

Card and Comer suggest that two fundamental mistakes contribute to the failure of reuse programs:

1. Organizations treat reuse as a technology-acquisition problem rather than a technology-transition problem (i.e., buying technology usually does not lead to extensive reuse).

2. Organizations fail to approach reuse with a business strategy.

The authors' experiences as promoters and supporters of reuse and as measurers of its effectiveness have lead them to believe that the overriding obstacles to reuse are economic and cultural, not technological.

Card views the reuse process in terms of an economic model of supply and demand. The model includes producers, consumers, and a distribution mechanism. Both producers and consumers develop software to meet specific, yet different, needs. To date, however, more emphasis has been on distribution mechanisms (i.e., classification schemes and repositories) rather than meeting consumer needs.

Within an economic model of reuse, reuse-oriented business goals should maximize the amount of software that goes from a producer's project to a consumer's project. The amount of software that producers are able to transfer depends upon the distribution mechanism, but more importantly, on how well their products match what the consumers want. The following factors affect this match:

- The quality and reusability of the producer's software
- The skill and knowledge of the consumer about reuse and the reusable software
- The degree of congruence between the producer's and consumer's requirements

Effective reuse must address all three of these factors, but Card focuses on the last factor, (i.e., how well the reusable components meets the consumer's requirements).

Alignment of requirements can be achieved most easily within a domain. Domain analysis techniques have helped to identify areas for reuse for both the producer and consumer. Alignment of requirements can also be achieved by looking beyond the current project, to future projects, anticipating and planning for

needs (i.e., tools, techniques and training). This approach requires an organization to develop repeatable processes and products within a market-driven business strategy.

In addition to the economics of reuse, four cultural issues have an effect on reuse:

1. Training
2. Incentives
3. Measurement
4. Management

Because reuse is a business strategy, it requires training at higher levels of management than are customarily involved in new technology adoption. Monetary or recognition rewards tied to the production and consumption of reusable assets can actively promote reuse within an organization. Management must support subsidizing production or other incentives to feed the needed cultural change that reuse requires. In addition, organizations need a mature, systematic development process with strong configuration management and quality assurance in order to effectively leverage reuse. These operations support the requirements for the Capability Maturity Model, Level 2. Card recommends that reuse should also be part of an organization's overall process improvement program.

[DIS95] Defense Information Systems Agency Center for Software, DoD Software Reuse Initiative, "Software Reuse Business Model (SRBM)," Technical Report, January 31, 1995.

This document describes a Software Reuse Business Model (SRBM) that was developed by the U.S. Army Space and Strategic Defense Command (USASSDC), Software Engineering Division at Huntsville, AL for the DoD Software Reuse Initiative, Defense Information Systems Agency, Center for Software. The SRBM is driven by the DoD Software Reuse Vision and Strategy and leverages previous work by other Government and commercial organizations. Its thrust is to steer away from re-inventing software to a new way of constructing software by reusing domain-specific architectures.

Written from the acquisition perspective, the SRBM incorporates reuse principles into the acquisition cycle of software-intensive systems within DoD organizations. The SRBM emphasizes systematic reuse and defines the detailed steps to apply reuse principles. The SRBM consists of the following components:

1. Specific activities for defined user roles (e.g., User, Domain Manager, and Program Manager)
2. Information needed for the activities of each role

3. Required policies, procedures, standards, and guidelines required

4. Tools, techniques and methodologies

The audience for the SRBM is DoD managers and practitioners. The end users are the Domain Manager (DM), the Program Manager (PM), and Support Staff (contracting personnel, finance personnel, technical staff and legal counsel). Other end users are Project Executive Officers (PEO) or the Designated Acquisition Commander (DAC).

The SRBM provides a formal structure to formulate a reuse business strategy while considering the viability and applicability of reuse. The DoD developed a general business model which each service branch will implement in its own way. There is no single reuse business strategy that is appropriate to every system acquisition. Using this generic model, an organization can tailor the SRBM and exploit the benefits of reuse while reducing risks and controlling costs. The SRBM can be exercised at any time during the system life cycle and provides a process to support decision-making.

The document describes the SRBM as a top-level model concept and then provides addition detail with IDEF 0 (Integrated Computer Aided Manufacturing (ICAM) Definition). Using the IDEF 0 notation, components are defined and decomposed as inputs into an activity, constrained by controls (policies, procedures and regulations) and by mechanisms (resources, guidance and tools), resulting in an output from each activity.

The SRBM provides and organization with the following mechanisms:

- A set of reuse business strategies that can be used to reduce the risk of implementing reuse
- Step-by-step instructions for performing cost and economic analyses
- A matrix table to determine common functionality among systems

The SRBM supports engineering activities, business planning and contracting activities related to software. The SRBM introduces a process for reuse-specific acquisition activities such as developing a domain infrastructure and then implementing and maintaining the domain. A Domain Manager must formulate a strategy appropriate for systems in a given domain.

To assist the Domain Manager, the SRBM defines several archetypes, or recurring patterns, of domains within Government and industry:

- Vendor-owned domain
- Government-support standard
- Value-added reseller
- Government-owned architecture

- Government-owned domain
- Re-engineering
- Public library
- Commercial library

The eight archetypes vary in the degree of control the Government has over reusable assets and their suppliers, the amount of reuse that can be expected, and the cost of exploiting reuse. These archetypes were derived from successful software reuse strategies and acquisition scenarios of the Reuse Acquisition Action Team (RAAT), Association of Computing Machinery (ACM) Special Interest Group for Ada (SIGAda) Reuse Working Group in March 1994.

Characteristics of a particular domain are analyzed by the Domain Analyst to determine which archetype applies. The SRBM defines selection criteria are used by the Domain Manager and the Domain Analyst to determine which archetypes are the most appropriate for the given domain. These selection criteria are budget, schedule, existing assets, expected uses in the domain life, commonality, variabilities, standardization and stability. Archetypes can be variants or hybrids formed from one or more archetypes. To characterize a domain, the Domain Analyst and Domain Manager must also consider other aspects of a domain such as programmatic, product, capability, and control.

A domain is evaluated by assigning values to the criterion as characterized by the organization's situation. Situational values for selection criteria are matched to those values for pre-defined selection criteria for each of the archetypes. Results are scored and the "best" fit and "second best" are matched. If criteria are not met, a type is not eliminated, but suggests risks may exist in those areas.

Selection of archetypes assists the Domain Manager in formulating business strategies. Organizations vary in the degree of investment, ownership, and control over domain assets between the Government and industry. Consequently, the SRBM has pre-defined strategies and preferred values for each of the archetypes to assist the Domain Analyst and the Domain Manager.

Within the SRBM is a Common Functionalities Table (CFT), a mechanism to determine common functionality within a domain of interest. The CFT helps to determine economic viability, implement a domain infrastructure, determine availability of components, perform reuse requirements analysis, and develop a reuse strategy. The CFT can also be an indicator for potential reuse. The CFT helps plan for the following scenarios:

- Which reusable components from one system might be reused in a system under development?
- Which reusable components from one system might be reused for a future system?

- What investments should be made in developing new reusable components?

The assessment mechanisms used in the SRBM flow down from many that exist in the software industry; the Domain Assessment Model, the Reuse Capability Model of the Software Productivity Consortium, the Reuse Strategy Model of STARS, the Software Capability Evaluation, and the Capability Maturity Model of the Software Engineering Institute.

Archetype selection and its associated domain analysis can also identify business entities involved in the acquisition process within a domain. By determining the related business roles, an organization can assign responsibilities to each role and specify the contractual/financial relationships among them. The following business roles are supported by the reuse archetypes:

- Acquisition Executives
- Acquisition Managers
- Research and Engineering
- Contractors
- Vendors
- Library Operations Organizations

As defined in an accompanying appendix, these business roles have differing areas of responsibility in Application Engineering, Domain Management, Domain Analysis, Domain Implementation, and Application Engineering Support (for the reuser). The latter includes operating libraries and collection and maintaining assets. The SRBM also considers Contractual/financial relationships between business entities including source funding (e.g., IR&D, or program dollars), the type of contract (e.g., Fixed Price or Cost Plus Fixed Fee), and usage rights (e.g., unlimited or restricted).

Not only can the Domain Manager analyze his domain and plan a business strategy, but he also can plan for asset production with the SRBM. His plans for asset production must also consider the cost and benefit of reuse. The SRBM provides a Cost and Economic Analysis that consists of reuse impact concepts and techniques. Data for the Cost and Economic Analysis is collected from completing questions with a domain perspective and domain emphasis. Since reuse is not yet widespread throughout the industry, the model does not provide benchmarks for comparisons after the analysis is completed.

In addition to Cost and Economic Analysis methods, the SRBM identified the following reuse metrics:

- Cost avoidance
- Return on investment (savings/cost)

- Productivity metrics
- Equivalent new source statements
- Quality metrics
- Schedule metrics
- Product development productivity (PDP) - KSLOC (Thousand source lines of code) size of product/effort
- New code productivity
- Code reuse productivity

The authors of the SRBM have plans to improve and enhance the current SRBM. Future work is anticipated in the areas of usability and transfer of the model into practice.

[DOD94a] DoD Software Reuse Initiative (SRI), Technology Roadmap, Version 2.0, Volume 1: Technology Assessment, October 4, 1994.

This report is in response to the DoD Software Reuse Vision and Strategy authored by Linda Brown as described above in [BRO92]. The DoD Vision and Strategy calls for "a technology-based investment strategy which identifies, tracks and transitions appropriate reuse-oriented process and product technologies." Critical technologies were identified within the context of domain engineering and application engineering. By evaluating critical technologies and their maturity profiles, an organization can judge which lagging technologies are recommended for investment within a ten-year time frame. Investment recommendations need to determine whether a technology will be accelerated by investment alone. In some cases, other factors or dependencies may impede progress regardless of funding, so that the potential return is not worth the investment at a given point in time.

The report identified software reuse as a critical technology and that to reach maturity, other supporting software reuse technologies will also need to mature. Two additional critical technologies that facilitate software reuse are reuse-based application engineering and domain engineering. Enabling technologies for reuse-based application engineering and domain engineering are representation, process modeling, composition and generation, language mechanisms, libraries and repositories, methods, software engineering environments, reengineering, and measurement and assessment. When software reuse fully matures, it will be an integral part of software engineering, and will disappear as an independent concept

The DoD technology roadmap was developed in the context of previous work done by Redwine and Riddle found in "Software Engineering Technology Transfer Practices, International Perspectives in Software Engineering," January, 1993. Redwine and Riddle found that the average time frame for an engineering technology to mature is 15-20 years. Within this time frame, Redwine describes

Milestone 0 as initially marked by the appearance of a key idea underlying the technology and a clear articulation of the problem; whereas Milestone 4 is marked by substantial evidence of value and applicability. Widespread use can require another decade.

The assessment concluded that software reuse has been developing bottom-up, consistent with how engineering disciplines have developed in the past. This development process starts with code modules, then expands to design and requirements, and architectures in domain analysis. As an engineering discipline matures, it acquires engineering handbooks, standard notations, objects, tools, and then it is taught to students of the discipline prior to widespread use.

On March 30, 1995, an updated version of this document was published by the DoD as Version 2.2 and reflects review comments.

[DOD94b] DOD Software Reuse Initiative (SRI), Technology Roadmap, Version 2.0, Volume 2: Implementation Plan, October 4, 1994.

This document presents a strategy and plan for additional DoD investments in reuse technology and flows down from the SRI's Volume 1: Technology Assessment described above. Major programs and institutions included in the strategy and plan include ARPA STARS, DSSA, SEI, SPC and the DISA Software Reuse Program. The strategy and plan covers a five year time frame from FY 1996 through the year 2000.

Five thrusts were identified in the strategic plan:

1. Mathematical foundation for reuse
2. Framework for Measurement and assessment
3. Domain and application engineering
4. Process Modeling
5. Integrated Environment Testbed

Included in the first thrust of Mathematical foundations for reuse was component certification using formal methods.

Formal methods use systematic mathematics to specify, develop and verify systems. A formal method provides notations and processes that enable a specification to be described rigorously and unambiguously. Software can be developed from formal specifications and the transformation from specification to software can be verified mathematically. Formal methods can be applied to safety critical systems.

Formal methods have potentially significant relevance to software reuse. Formal methods enable precise and concise specification of software and relations between components. Formal methods provide proof of the correctness of software

and consequently, the authors feel, can be useful in the production and validation of reusable assets.

However, the state of the practice of formal methods is fairly immature. In general, it is a topic for academia, research and Government. Formal methods are perceived as not readily scaleable upward from experiments to industrial-size applications. Formal methods are difficult and time-consuming to apply to an entire system over its life cycle. Some formal methods support particular phases in the development life cycle better than others. The methods, themselves, can be immature and lacking in automated support. Hybrid approaches applied to portions of a large domain, using both manual and automated tools, are a compromise often sought. Systems can be partitioned, formal methods used on the partitions, then the partitions re-integrated later. Formal methods may be used to some degree in the U.S. private sector, but are more common in Europe, especially in the United Kingdom.

Some gaps remain in formal methods. The link supporting specifications and code needs to be strengthened. Formal methods need to accommodate larger inference steps and the ability to specify non-functional behavior (e.g., reliability, safety, and performance). Supporting standards need to be developed, as well tools that adequately scale upward.

Both Volume 1 and Volume 2 of the Technology Roadmap for DoD Software Reuse Initiative (SRI) were developed by MITRE and the University of Houston under the direction of Mr. Don Reifer. The staff extracted information from experience, literature survey, data gathered from software engineering managers and researchers.

On March 30, 1995, an updated version of Volume 2: Implementation Strategy was published by the DoD as Version 2.2 and reflects review comments.

[JAW90] Jaworski, Allan, Fred Hills, Thomas A. Durek, Stuart Faulk, John E. Gaffney, "A Domain Analysis Process," Interim Report 90001-N (Version 01.00.03), Software Productivity Consortium (SPC), Herndon, VA, January 1990.

As part of his domain analysis process, Jaworski established the relationship between productivity enhancement through reuse versus the cost of reuse. As illustrated in Figure A-1, he defined the parameters that indicate when reuse pays, and when it does not.

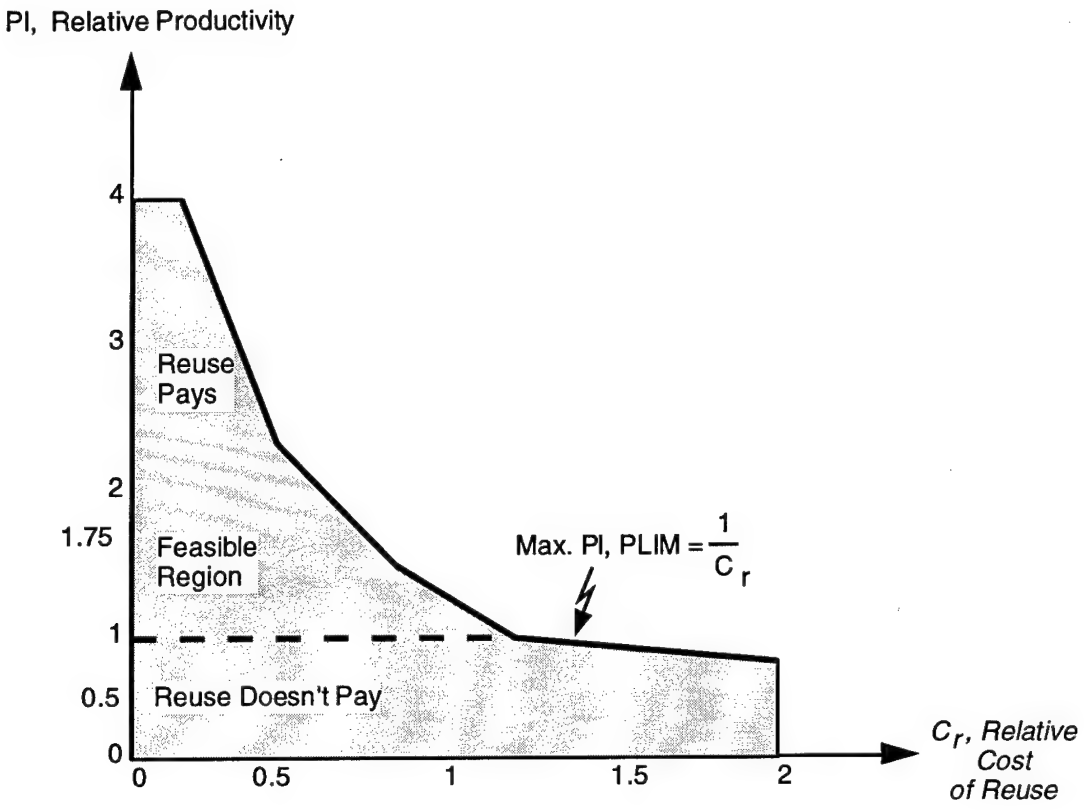


Figure A-1. Productivity enhancement through reuse versus cost of reuse [JAW90]

He also developed a basic software reuse equation as shown in the following equation:

$$C_{us} = C_{un} - (C_{un} - C_{ur}) \bullet R$$

where

C_{un} = average cost per unit of new software in the product
S_n = amount of new software in the product
C_{ur} = average cost per unit of reused software in the product
S_r = amount of reused software in the product
C_{us} = average cost per unit of the software product
S_s - size of the software product
R = proportion of code reused = S_r / S_s; $1-R = S_n/S_s$

The above equation applies to the software product developed for an application system. The unit costs, C_{un} and C_{ur}, cover all the activities required to create the new and reused code in the system. Jaworski ties his software reuse to

domain analysis by constructing the following equation:

$N_o = C_d/(1-C_r)$.

where

N_o is the required number of uses for an investment in domain engineering to break even

C_d is the ratio of the cost to produce a unit of reusable code to the cost per unit to engineer the code for a single application

C_r is the ratio of the cost to adapt a unit of reusable code to the cost of implementing a new unit of code.

Even if C_d is relatively small, a high value of C_r can make domain engineering unprofitable. However, if we can significantly reduce C_r by automating software reuse, we can achieve a break even and even higher levels of productivity. Therefore, effective domain engineering becomes critical to the economics of reuse.

Jaworksi cites Gaffney in his work in the late 80's as supporting his premise. Gaffney provides a mathematical formulation of the issues associated with amortizing the costs of domain analysis, design and implementation efforts across multiple projects. Jaworksi provides an appendix to his report that consists of a detailed checklist of questions that can be used to determine the feasibility of domain analysis within an organization.

[JON95] Jones, Capers, "Return on Investment in Software Measurement," Proceedings of the 6th International Conference on Applications of Software Measurement (ASM), October 30-November 2, 1995, Orlando, FL, pp. 349-427.

Capers Jones identified the following top five technologies in terms of Return On Investment (ROI):

1. Full software reusability
2. Fully integrated I-CASE tool suites
3. Winning a Baldridge award
4. Software quality measurement
5. Software cost and quality estimation tools

Using some of these technologies can quickly result in a return on investment within 3, 6, 12, or 24 months. However, the most significant returns on reuse usually result in a longer time frame (i.e., 36 months). Jones points out that software reuse can include architectures, estimates, plans, requirements, designs, code, user document, human interfaces, data and test cases.

Jones has approximated the return for each dollar invested in reuse over varying time periods as shown in the Table A-1 below.

Table A-1. Approximate return for each dollar invested in reuse

Types of reuse	12 months	24 months	36 months	48 months
Full Software Reusability	$1.00	$3.00	$15.00	$30.00
Reusable Architectures	$0.00	$0.20	$0.75	$1.50
Reusable Estimates	$0.20	$0.30	$2.00	$3.00
Reusable Plans	$0.15	$0.25	$1.00	$2.00
Reusable Requirements	$0.10	$0.40	$1.50	$3.00
Reusable Designs	$0.10	$0.40	$2.50	$5.00
Reusable Code	$0.15	$0.50	$2.50	$6.00
Reusable User Documents	$0.05	$0.10	$0.75	$1.50
Reusable Human Interfaces	$0.00	$0.15	$0.50	$1.00
Reusable Data	$0.20	$0.30	$1.75	$3.50
Reusable Test Cases	$0.05	$0.40	$1.75	$3.50

Naturally, cumulative reuse across multiple types of reuse is the most beneficial; however, the most sizable payoffs across types occur after 36 and 48 months. Reusing designs and code are worthy of investment, but an organization must be aware that these types of reuse have their best payoffs in the out years. If a near term return is desired, it may be useful to plan investments in other types of "non-traditional" reuse.

[LIM94] Lim, Wayne C., "Effects of Reuse on Quality, Productivity, and Economics," ***IEEE Software,*** **September 1994.**

Wayne Lim documents metrics from two case studies at Hewlett-Packard (HP) that demonstrate improved quality, increased productivity and reduced time-to-market. He applied his metric models to case studies in the area of Manufacturing Productivity in the Software Technology Division and in the San Diego Technical Graphics Division, both of HP.

Lim points out that increased productivity from reuse does not necessarily shorten the time-to-market. To reduce the time-to-market, reuse must be used effectively on the critical path of a development project, that is, the chain of activities that determine the total project duration.

By conducting a reuse assessment, he cites the following findings:

1. Quality - Because work products are used multiple times, the defect fixes from each reuse accumulates, resulting in higher quality. Reuse provides incentives to prevent and remove defects earlier in the life cycle because the cost of prevention and debugging can be amortized over a greater number of uses.

2. Productivity - Reuse improves productivity because the life cycle now requires less input to obtain the same output. Reuse can also improve a product's maintainability and reliability, thereby reducing maintenance labor costs.

3. Time-to-market - A reduction of up to 42% of calendar months was shown in Lim's case studies.

He cautioned, however, that software reuse is not free. Reuse requires resources to create and maintain reusable work products, a reuse library and reuse tools. He found that the relative cost of creating a reusable code components is about twice that of creating a non-reusable version. The cost to integrate reused components into new products ranged from 10-20% of the cost of creating a non-reusable version. Lim used the well-established net-present-value method of economic analysis and variations across projects were due to domain differences.

The most significant increase in labor cost for reuse occurs in the investigation and external design phases. This is because the producer of the work product requires a greater amount of time to understand the multiple contexts in which the work product will be reused. Nonetheless, Lim believes that the results of economic cost-benefit analyses indicate reuse can provide a substantial return on investment in the long term.

[MOS95] Moseman, Lloyd K., "Software Development: Quo Vadis?" *Crosstalk*, November/December 1995, Volume 8, Number 11, p. 2-3.

In his farewell address, Mr. Lloyd Moseman, Deputy Assistant to the Secretary of the Air Force, provides a perspective of where we've been and where we're going in software development. In 1990, he said that if the 1990s reveal a silver bullet, that bullet will be reuse. Now in 1995, he feels that his statement may have been naive. To date, reuse has not been practiced on any major scale, and he feels that the foundation of software engineering practice and software process maturity were prerequisites to reuse on a major scale. He believes that before the year 2000, the potential for major benefits from reuse will arise.

Moseman believes that architecture-based product lines show the greatest promise for reuse. He believes that for each functional domain, there must be an architecture with engineered qualifications, stature, and the role of the software architect. These architects should not be in the Government, but the Government will need to play a role by fostering the establishment of product lines from these architectures.

Product lines go beyond software technology in that management must plan for capital investment. Investments must be made in hiring the architect, architecture creation, reusable component development, and pre-certification of Commercial-Off-The-Shelf (COTS) components. The product line paradigm moves away from traditional development that focuses on specific requirements with funding and management that is oriented to single systems or projects. The contractor with an effective project line will win competitions because of lower cost, higher quality, quicker delivery, and predictable performance.

[PYS92] Pyster, Art, "Reuse Adoption Guidebook," Software Productivity Consortium, SPC-92051-CMC, Version 01.00.03, November 1992.

Pyster defines a reuse adoption process as a set of activities to incorporate the practice of software reuse as a permanent part of an organization's culture and way of doing business. More simply, the reuse adoption process is a way to institutionalize reuse. The reuse adoption process is derived from a broad base of experience and research in the areas of reuse, technology transfer, planning, risk management, process improvement, and economics. Generally, it is a technology transfer process for an organization, but specialized for transferring reuse technologies (i.e., processes, methods, and tools). The defined process can also be used to improve an organization's current reuse practices to obtain better results.

Pyster's guidebook uses SADT (Structured Analysis and Design Technique) diagrams to define the reuse adoption process. Detailed activities are identified with inputs, outputs, controls, mechanisms, and roles. Pyster defines the following top level activities:

1. Initiate Reuse Program Development
2. Define Reuse Program
3. Analyze Reuse Adoption Strategies
4. Develop Reuse Action Plan
5. Implement and Monitor Reuse Program

Each of these high level activities are further decomposed into their lowest level of tasks. Some of the roles performing these tasks (i.e., sponsor, reuse champion, reuse agent, user) may be found in the existing organizational structure in the persons comprising Software Engineering Process Groups. To support the

reuse process, these roles must be fully integrated into the activities of software definition.

Planning through assessment is a key concept that runs through the reuse adoption process. Both reuse capabilities and business area potentials need to be assessed and a reuse plan developed around them. If the potentials for both are high, then an organization can justify the investment necessary to improve its reuse capability to maximize this high potential. However, if the potential is limited, then it may not be cost-effective to greatly improve an organization's reuse capability in a chosen business area.

Pyster points out that reuse is often narrowly viewed as a technique that, if applied, will reduce cost, neglecting the fact that adopting and performing reuse is not free. The costs associated with reuse are an investment and the benefits gained from reuse are the return on this investment.

[RAT94] Rathbun, Robert W. "Software Reuse Metrics," Proceedings of the Software Technology Conference (STC), Volume 1, Salt Lake City, UT, 1994.

Rathbun believes the only true reuse issue is the way the project manager or technical lead plans for and manages reuse within a project's software development and maintenance. Within the software reuse process, he measures reuse activity for the following reasons:

1. It enables understanding of the reuse process.
2. Software reuse in projects can be planned and controlled.
3. The best application of effort and resources can be determined.
4. The quality of the artifacts of development and maintenance can be evaluated.
5. The benefits of software reuse can be evaluated.
6. The software reuse program can be improved.

Rathbun recommends formalizing a measurement plan and applying it to pilot projects. He constructed a mathematical model for project-based cost avoidance and return on investment and demonstrated their use with sample data.

Rathbun's work is based the DoD Software Technology Strategy published in 1991 and the DoD Vision and Strategy for Software Reuse of 1992 [BRO92] and has these high level goals:

- Reduce equivalent software life cycle cost by a factor of two.
- Reduce software problem rates by a factor of 10.

- Achieve new levels of DoD mission capability and interoperability via software.

These high level goals resulted in the current move away from reinventing software to a process-driven, domain- specific, architecture-centric, library-based way of constructing software.

These goals drove the creation of a ten-point strategy to reach these goals, one of which is devoted to the collection of software reuse metrics. The plan establishes procedures to collect metrics to measure the payoff from the reuse initiatives. It also aids the developers in the selection of reusable components. Rathbun cautions that reuse measurement is very immature, much more than the immature area of measurement, in general.

[SPC91a] Software Productivity Consortium (SPC), "Domain Analysis Workshop Presentations," SPC-91186-MC Version 01.00.00, September 26-27, 1991, Herndon, VA.

Sidney Bailin, in his presentation at the Domain Analysis Workshop in 1991, identified the following indicators that an organization needs to perform domain analysis:

1. A pressing need to streamline the development process by standardizing engineering practice
2. A perceived opportunity to reuse more than is being reused
3. A problem of attrition of expertise due to development personnel turnover

Bailin, like Jaworksi, shows a close tie between domain analysis and an organization's business strategies.

[VCO93] Virginia Center of Excellence for Software Reuse and Technology Transfer (VCOE) "Reuse Adoption Guidebook," Software Productivity Consortium, SPC-92051-CMC, Version 02.00.05, November 1993.

This Reuse Adoption Guidebook issued by the Virginia Center of Excellence for Software Reuse and Technology Transfer (VCOE) is an update to [PYS92] annotated above. In addition to including the definition and decomposition of the reuse adoption process, the updated guidebook provides assistance in performing domain assessment, reuse capability assessment, reuse adoption, and strategy development. The guidebook also has appendices describing reuse adoption risks, and assessment of legal and contractual reuse issues. Similar to the reuse adoption process, each of these assessment techniques are defined by SADT analysis.

The domain assessment method supports the definition of a reuse program by understanding the reuse context and assessing the reuse potential. The purpose of the SPC's domain assessment is to understand the potential for reuse in an organization and to help determine how much to invest in reuse and where to focus the investment. Prior to performing a domain assessment, the organization must commit the resources required to perform the domain assessment. The following tasks are included in the domain assessment:

1. Organize the domain assessment team.
2. Identify specific product domains to assess.
3. Assess domain factors.
4. Develop assessment findings.
5. Develop supporting material.
6. Report domain assessment findings.

The inputs to the process of domain assessment are the organizational profile, current reuse situations, product plans, marketing information, existing assets, product family requirements, domain history, technology trends, and standards. The outputs are domain assessment findings, supporting material, findings presentation and a domain assessment report. Controls are an organization's reuse adoption objectives. The mechanisms needed to implement the domain assessment are the domain assessment model and domain experts. The exit criterion is the review and approval of the domain assessment findings and supporting material by the sponsor.

In addition to the domain assessment method, the reuse capability assessment method also supports the definition of a reuse program by understanding the reuse context and assessing the reuse potential. The purpose of the reuse capability assessment is to gain an understanding of an organization's process with respect to reuse sufficient for planning improvements (i.e., identifying process strengths and improvement opportunities). Prior to performing a reuse capability assessment, the organization must commit the resources required to perform the reuse capability assessment. The following tasks are included in the reuse capability assessment:

1. Organize the reuse capability assessment team.
2. Identify the process to assess.
3. Assess the organization's process.
4. Develop assessment findings.
5. Report reuse capability assessment findings.

The inputs to the process of reuse capability assessment are the organizational profile, the current reuse situation, the organization's process, methods, tools, structure, and skills. The outputs reuse capability assessment findings, findings

presentation and a reuse capability assessment report. Controls are an organization's reuse adoption objectives. The mechanisms needed to implement the reuse capability assessment are the reuse capability model and the reuse capability assessment team. The exit criterion is the review and approval of the reuse capability assessment findings and supporting material by the sponsor.

As part of the reuse adoption process, the SPC provides a guide to developing a reuse adoption strategy. The purpose of the reuse adoption strategy development is develop a strategy to meet the established reuse adoption goals and objectives. Prior to developing a reuse adoption strategy, an organization must establish reuse adoption goals. The following tasks are included in the reuse adoption strategy development:

1. Develop the product approach.
2. Develop the business model.
3. Develop the process approach.
4. Development the organizational approach.
5. Develop the environment approach.
6. Develop the transition approach.

The inputs to the process of reuse adoption strategy development are the organizational profile and supporting materials. The output is the reuse adoption strategy. Controls are an organization's reuse adoption objectives, reuse adoption goals, and constraints. The mechanisms needed to implement the reuse adoption strategy development are reuse agents. The exit criterion is demonstration that the reuse adoption strategy addresses all strategy components.

For all of these three processes (e.g., the reuse adoption process, the reuse capability assessment process and the process for reuse adoption strategy development), this updated guidebook provides very detailed descriptions as to how to perform each. In addition to providing a textual descriptions, the guidebook provides diagrammatic views of the techniques, annotated outlines, and worksheets.

Appendix B - Annotated Bibliography for Domain Analysis

The information gleaned from this literature survey of domain analysis was used to determine the operational context of the Reuse Context for Asset Quality Certification and to assess the impact of this previous research on the development of the Certification Framework.

The annotations in this appendix summarize the essence of each of the referenced publications. Summaries vary in length; those that are longer provide additional details because the reference appeared to be a flagship among others. The shorter annotations were still included to serve as a pointer to the complete reference if more details are of interest.

This annotated bibliography this area is not exhaustive, but gives a flavor of the previous research that has been accomplished. Some of these references were used in other appendices.

[ARA89] Arango, G. "Domain Analysis - From Art Form to Engineering Discipline," *Proceedings of the 5th International Workshop of Software Specifications and Design*, p. 152-159, 1989.

Arango proposes a different approach to domain analysis called "practical domain analysis." He maintains that pure domain analysis is a theoretical problem associated with scientists and systems analysts and feels the need to further develop domain analysis into a practical activity. Practical domain analysis methods are based on a view of reusers as learning systems.

He has written this paper to advance a conceptual framework for practical domain analysis that is at a meta-level rather than another particular method for domain analysis. Arango believes that different domain analysis methods may not be comparable; each has been designed for reuse in different situations.

Arango believes that for domain analysis to become a practical technology, the following three activities need to happen:

1. Understand the conceptual foundations of the process.
2. Produce an unambiguous definition using specification techniques.
3. Provide adequate support tools.

Completing these activities moves reuse away from an "art form" and toward and engineering discipline.

Arango's view for practical domain analysis is seen as a systematic evolution of a reuser's model of the domain, attaining and maintaining a desired level of performance. Practical domain analysis answers the question of how to model a domain that is incrementally constructed and evolved to achieve a specified level of performance with a given target reuser. Arango explains that the number of specifications covered by a reuser is potentially infinite. In practice, performance properties are measured over a selected set of sample specifications, and a set of benchmarks result. Reuse benchmarks are designed to reflect patterns of reuse in the environment of a system, based on such properties as how recently the component was reused, frequency of reuse over some period of time, or some measure deemed relevant to the purpose of the study.

The goal of practical domain analysis is to find a systematic method to identify information in the problem domain which, if available to the reuser in an appropriate form, would attain a specified level of performance. The analyst captures the information identified as relevant and evolves the acquired information to enhance or maintain the performance of a reuser. Arango applies his concept to an example, the GTE Assets Library System, at GTE Data Services.

[BIG88] Biggerstaff, Ted. J. , "The Nature of Semi-Formal Information in Domain Models," Technical Report STP-289-88, Microelectronics and Computer Technology Corporation, Austin, TX, September 1988.

In his discussion of semi-formal information in domain models, Biggerstaff describes several levels of components, spanning code to conceptual abstractions:

- Code-execution of instructions - Is implementation-specific and constrained by programming language, formal objects and operational form.
- Software engineering design - Weakly related to informal concepts, is implementation-specific, constrained by language and application domain, has semi-formal objects, is abstractly operational and presents system in reduced detail, abstract away detail.
- Generalized software engineering design - Weakly related to informal concepts, provides widely reusable designs
- Conceptual abstraction - Strongly related to informal concepts, not implementation-specific, has object-like structure, has non-operational or prescriptive form.

These differing levels of abstraction are listed from lowest to highest and can be applied to constructing domain models during domain analysis.

[BOR84] Borgida, Alexander, John Mylopoulos and Harry K.T. Wong, "Generalization/Specialization as a Basis for Software Specifications," *On Conceptual Modeling*, pp. 87-117, Springer-Verlag, New York, NY 1984.

Borgida believes that in conceptual modeling, generalization should be used as the cornerstone in designing data-intensive applications. He suggests that the best path for success is to create systematic and structured descriptions of highly detailed world models where each concept has variations.

[BRU88] Bruns, Glen and Colin Potts, "Domain Modeling Approaches to Software Development, " Technical Report, STP-186-88, Microelectronics and Computer Technology Corporation, Austin, TX, June 1989.

Bruns purports that domain modeling is a pervasive activity that includes domain analysis. He shows the relationship between five design approaches and domain modeling. He evaluated each of the design approaches with respect to modeling primitives, domain analysis, analysis/validation of the domain model, and its specification and implementation.

[COH89] Cohen, Joel, "Software Reuse for Information Management Systems," Position Paper of the Reuse in Practice Workshop, Software Engineering Institute (SEI), Pittsburgh, PA, July 1989.

Cohen wrote a position paper motivated by the goal to reduce the cost of building a complex imagery information management system. For these systems, a domain model was constructed, a generic architecture was developed, a classification scheme for string reusable components was defined, and a library was populated. Cohen used the domain analysis method of Prieto-Diaz and Gish.

[COH92] Cohen, Sholom, Jay L. Stanley, Jr. A. Spencer Peterson, Robert W. Krut, "Application of a Feature-Oriented Domain Analysis to the Army Movement Control Domain," Technical Report, CMU/SE 1-91-TR-28 ESD-TR-91-28, June 1992.

Cohen applied the Feature-Oriented Domain Analysis (FODA) method to the window manager domain to validate the approach for its future use. During his application, he operated under constraints since commonalities in the domain were neither well-understood nor well-documented at the time of his study. No domain expertise existed before the analysis, and no user was available to test the results.

Cohen learned that the following activities must be performed for successful domain analysis.

- Clearly define the users - address needs, elicit requirements for software implementation and system interfacing
- Identify domain experts early in the process
- Construct an enactable model
- Establish a community of interest
- Provide support for domain experts

Cohen also found that domain analysis can be used to improve communications.

[DOD95a] Department of Defense, "Software Reuse Symposium," March 23, 1995, Huntsville, Alabama.

Two papers at the Software Reuse Symposium discussed domain analysis; one addresses domain modeling, the other a tool for domain modeling. First, the Organization Domain Modeling (ODM) method of domain analysis was developed by Unisys and Organon Motives under the STARS program. ODM is a prescriptive domain analysis method which directly relates to the STARS Conceptual Framework for Reuse Processes (CFRP).

ODM consists of a process and work product model that can be instantiated in a variety of sequences and project structures. Using process trees, ODM integrates the business and technical aspects of domain modeling. ODM guides the selection of strategically appropriate domains. Within the domain of focus, ODM guides the analysis of existing and envisioned domain capabilities culminating in a

specification for a set of reusable assets. Consequently, ODM directly supports reuse planning and domain selection. This paper documents the ODM method as applied at several sites; Hewlett-Packard, Unisys and to the AF Comprehensive Approach to Reusable Defense Software (CARDS), and the Tomahawk missiles program.

KAPTUR, a domain engineering tool, was also presented at the Software Reuse Symposium. This public domain tool was developed by CTA and was primarily funded by NASA. KAPTUR supports side-by-side analysis of multiple systems in a domain by using different views of the same information. It runs on a Sun SPARCstation and commercialization is planned.

[DOD95b] Department of Defense, "Domain Scoping Framework", Version 3.1, Volume 2: Technical Description, 29 September 1995.

The DoD published this document describing a framework that addresses issues associated with how to define domains and product lines, how domains relate to each other, how to identify all DoD domains, how to establish product lines, and how to exploit the commonality among software systems within these domains and across domains.

The purpose of the framework and accompanying usage guidelines is to provide a basis for finding and exploiting maximum commonality among software systems as a means of improving the engineering of DoD software. Volume 2 of this two part set provides details of the framework for technical users; whereas, Volume 1 is a shorter document, intended for executives and managers, that briefly explains the framework and elaborates more on its broader context and its benefits. An appendix to Volume 2 identifies outstanding issues that will be addressed in future versions.

Volume 2 assumes that a domain-specific approach to software engineering is a foundation for good engineering practice and promotes natural reuse, standardization, increased quality, and reduced cost. Related to a domain-specific approach is the establishment of product lines as a business area within an enterprise as a means of exploiting its knowledge and experience.

Elements of the framework include common definitions and a set of factors for characterizing domains and making decisions. The following terms and their definitions were chosen from standards in the industry:

- Business area - A coherent market characterized by (potential) customers possessing similar needs.
- Domain - A distinct functional area that can be supported by a class of software systems with similar requirements and capabilities. A domain may exist before there are software systems to support it.
- Domain boundary - A frame of reference for an analysis (i.e., the set of constraints that represent what is part of the analysis and what is

outside the analysis). The domain boundary may change as more knowledge about the domain is gathered.

- Product line - A collection of products (existing and potential) that address a designated business area.
- Framework - A structure for supporting or enclosing something, especially, a skeletal support used as the basis in something being constructed; a basic arrangement, form, or system.

This document defines eight factors for characterizing domains and making decisions and organizes these eight factors into two groups:

Domain Profile Factors

Domain Identify - High-level or defining characteristics of the domain, and location in domain taxonomy

Functional System Requirements - Dominant functions or features of systems in the domain

System Characteristics - Dominant subsystems, characteristics, constraints, and nonfunctional requirements of systems in the domain

Software Characteristics - Dominant characteristics of software subsystems that support the domain

System Deployment - Where systems in the domain are deployed

Decision Support Factors

Domain and Organization Assessment - Extent to which existing domain knowledge and experience, software assets, homogeneity, maturity provide reuse opportunities

Market Assessment - Potential for making profitable use of domain knowledge

Resource Constraints - Organization or enterprise limits on extent of analysis and engineering possible

The framework and these factors can be applied within an organization or across organizations. A Goal-Question-Metric (GQM) approach is used to specify the information to be collected about the factors. A common set of scenarios and their relation to MIL-STD-498 is described relating the framework to an understood process. The framework was applied to two example domains, the Joint C^4I Global Command and Control System (GCCS) and a Navy Program Executive Office (PEO).

[FAC94] **Facemire, J. Jeff , Aleisa Petracia, and Stephen Riesbech, "Software Architecture Seminar Report," Software Technology for Adaptable Reliable Systems (STARS), Central Archive for Reusable Defense Software (CARDS), Informal Technical Report, Contract No F19628-93-C-0130, January 29, 1994.**

Facemire defines domain engineering as the systematic identification of commonalities among a group of related software systems. Domain engineering is composed of domain analysis, domain design and domain implementation. Facemire defines domain engineering products as a domain model, domain specific software architectures, and domain design classification terms. Facemire feels that by using these products, asset production can be focused on reuse.

[GIL89] **Gilroy, Kathleen, Edward Comer, J. Kaye Grau, Patrick Merlet, "Impact of Domain Analysis on Reuse Methods," Final Report CO4-087LD-0001-00, U. S. Army Communications - Electronics Command, Ft. Monmouth, NJ, November 1989.**

Gilroy introduced an object-oriented concept of adaptation analysis, that is, the identification of differences among application systems. She proposed that the activities of successful domain analysis are modeling the domain, architecting the domain, and developing software component assets.

[HES90] **Hess, James A. , William E. Novak, Patrick C. Carroll, Sholom G. Cohen, Robert R. Hollbaugh, Kyo C. Kang, A. Spencer Peterson, "A Domain Analysis Bibliography," Carnegie-Mellon University and the Software Engineering Institute, Special Report, CMU/SEI-90-SR-3, 1990.**

Hess and his colleagues compiled a significant bibliography of references on domain analysis. Hess' bibliography was developed as part of the Domain Analysis Project at the Software Engineering Institute (SEI) and provides a historical perspective and background for further research.

Domain analysis literature has grown in the last 20 years and continues to grow. Entries in bibliography start in 1975, showing an initial growth spurt in 1985 and increased proliferation in 1988-1989. Those authors that can be considered experts in the field (i.e., publishing three or more articles during this period) are Arango, Bailin, Batory, Lubars, Neighbors, Parnas, Prieto-Diaz. The following projects or companies were referenced in three or more listings: CAMP, CTA, DRACO, GENESIS, GTE, IDeA MCC, NASA, RLF, SEI, STARS, UC-Irvine, Unisys, UT-Austin.

Hess defines a domain as a set of systems which share common capabilities. Domain analysis as a process to identify and represent the relevant information in a domain. To perform domain analysis, information is derived from a study of

existing systems, knowledge is captured, with underlying theory and emerging technologies. Hess points out that gray areas exist within domain analysis; there is overlap into the disciplines of formal specification and representation of knowledge. Hess believes that domain analysis is the foundation for establishing a reuse program within an organization.

[HUT88] Hutchinson, J.W. and P.G. Hindley, "A Preliminary Study of Large Scale Software Reuse," *Software Engineering Journal*, Vol. 3, No. 5, 1988, pp. 208-212.

Hutchinson performed a domain analysis study of existing software using a method called commonality. His approach is to determine if components were reusable either as is, with modification, or not at all. He used a catalogue scheme to organize and retrieve components. His study consisted of a small number of components, yet served to test his ideas. A lesson learned was that he had inadequate cataloguing schemes and developing these proved to be more time-intensive than planned.

[ISC88] Iscoe, Neil, "Domain-Specific Reuse: An Object Oriented and Knowledge-Based Approach" in Will J. Tracz, *Software Reuse: Emerging Technology*, pp. 299-308, IEEE Computer Society, Washington, D.C., 1988.

Neil Iscoe's approach to domain-specific reuse is based upon object-oriented and knowledge-based technologies. He defined the following nine steps for prototyping using domain analysis techniques:

1. Create a domain model.
2. Implement the model.
3. Instantiate the system for the library domain.
4. Specify and generate programs within the domain.
5. Instantiate the system for another related domain.
6. Refine the model.
7. Compare the instantiations.
8. Identify the characteristics and traits that generalize across domains.
9. Identify the algorithms and techniques that can be used across a class of domains.

Using these techniques, he prototypes two systems; one for reconfigurable databases and another for microcomputer screens. His overall approach used an object-oriented style of structuring, a visually-oriented end-user interface, and a knowledge-based mechanism for transforming requirements into primitive

functions. Domain modeling and domain analysis played an important part on the success of his research.

[JAW90] Jaworski, Allan, Fred Hills, Thomas A. Durek, Stuart Faulk, John E. Gaffney, "A Domain Analysis Process," Interim Report 90001-N (Version 01.00.03), Software Productivity Consortium (SPC), Herndon, VA, January 1990.

Jaworski believes that domain analysis is the critical front-end activity associated with the SPC's Synthesis methodology. The Synthesis method of domain analysis is rooted in the prior work of Neighbors in 1984, Arango in 1988, Prieto-Diaz in 1987-1988, and Bailin in 1989. His thesis is that if standard high-level designs for software systems are developed, then they are not likely to change from implementation to implementation. These high-level designs can be routinely used as frameworks for structuring requirements and lower-level design knowledge. Over time, an infrastructure that supports reuse of software can be built. He sees that Synthesis is a process for software development that emphasizes the automated generation of software systems from software components and models designed for reuse. These artifacts and their relationships can be housed and best preserved as a reusable library of components.

Jaworski defines domain analysis as the software systems engineering discipline that identifies, organizes and models information in a problem domain to produce software requirements for a class of problems. He maintains that domain analysis is a subdiscipline of domain engineering which develops a stable requirements framework, serving as the basis of domain engineering efforts.

Jaworksi describes four steps in the domain analysis process using the Synthesis method:

1. Domain description - defines the scope, functional boundaries and terminology.
2. Domain qualification - analyzes the economic and technical feasibility of a cost-effective domain solution determined by the pre-defined boundaries.
3. Knowledge base creation - collects information into a series of modules to produce descriptions of what is necessary for implementation.
4. Canonical requirement development - creates a framework for requirements common to instances of the domain and applicable to potential variations.

The domain qualification of step 2 determines if there is a business case for performing domain analysis and other associated activities (i.e., sales forecast, economics tradeoff, and risk evaluation).

The knowledge base creation of step 3 consists of developing the following work products:

- Glossary
- Taxonomy
- Specification sheets from engineering data
- User manuals and scenarios
- Technical articles
- Mathematical relationships
- Basic engineering data
- Descriptions of the domain
- Relevant information from other fields

Jaworski's reference documents these work products from the domain analysis of the Satellite Operations Control Centers (SOCC) and could be used as a guide for other applications.

Jaworski believes that formalization of practices done in requirements and design activities is the principle subject matter of domain analysis. He defines a domain as a set of problems with similar requirements for which a common solution can be developed. Software domains are domains for which software systems are appropriate solutions and for which the appropriate solution may be a common family of software programs. Domain analysis can be thought of as part of the DoD 2167A System Requirements Analysis and Software Requirements Analysis. Reusable software components are produced in design and code phases after domain analysis.

The short term benefit of the Synthesis method is that software engineering becomes a repeatable discipline. The long term benefit is the building up of the knowledge and capability to automate the software engineering process and achieving large increases in software productivity. The goal of Synthesis is to reuse software requirements, design, code, test information, etc. across a domain in families of similar projects.

Domain analysis requires domain experts, system engineers and software engineers to understand the problem and its changes. The products of a domain analysis are a domain definition, a taxonomy, a feasibility analysis (i.e., "go/no-go"), a domain knowledge base, and a canonical requirements model. Domain analysis requires a team approach to the problem for successful products.

The development team's familiarity with the application domain is a significant factor in the success of reuse. The requirements and design phase accounts for 80% of the work; it is also the most difficult and cost costly to repair, as supported by Brooks and Boehm in separate works.

The SPC, the funding agency of the Synthesis method, has member companies in three application domains; control systems, signal processing and command and control. As such, these domains strongly influenced the direction of the examples and case studies used for the development of the Synthesis methodology. Jaworski's initial work in Synthesis was validated using data from the domain SOCC at NASA Goddard Space Flight Center (GSFC), Space Telescope and GOES I-M Control Centers.

[KAN90] Kang, Kyo C., and Sholom G. Cohen, James A. Hess, William E. Novak and A. Spencer Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical Report, CMU/SEI-90-TR-21, ESD-90-TR-222, November 1990.

Kang documents a feasibility study of Feature-Oriented Domain Analysis (FODA) as applied to a window management system. The study was performed by the Software Engineering Institute (SEI) in 1990 in this realistic domain, but was not considered exhaustive.

Kang describes FODA as a domain analysis method that represents commonalities among related software systems by identifying user-specific features of software systems in a domain. The method defines mandatory, optional and alternative system characteristics and describes products and processes and their associated technical issues. The range of features determines the customizable requirements.

FODA is based upon other domain analysis methods and helps to establish the proper scope to a problem, a critical component to success. FODA uses features, parameters, composition rules, behavior and functional views, rationale and issues/decisions.

FODA supports reuse at the functional and architectural levels of system development. Using FODA, a system can be modeled and its differences can be "abstracted away" from existing systems within the domain. FODA provides a generic perspective to developing new systems and results in a set of products that define a system within a domain, both its differences and similarities.

In addition to analyzing particular domains, the method can be used in communication, training, tool development and software specification and design. FODA provides additional value since it captures the thought processes used to develop software systems of a related class (i.e., domain expertise). A summary of the phases of the FODA method is shown in Table B-1.

Table B-1. A summary of the FODA method [KAN90]

Phase	Inputs	Process	Product	Description
Context Analysis	Operating environments, Standards	Context analysis	Context model	Environments in which the applications will be used and operated
Domain modeling	Features, Context model	Features analysis	Features model	End-user's perspective of the capabilities of the applications in a domain
	Application domain knowledge	Entity-relationship modeling	Entity-relationship model	Developers' understanding of the domain entities (objects) and their relationships
	Domain technology, Context model, Features model, entity-relation model, Requirements	Functional analysis	Data flow model Finite state machine model	Requirements analyst's perspective of the functionality of the applications
Architectural modeling	Implementation technology, Context model, Features model, Entity-relation model, Design information	Architectural modeling	Process interaction model Module structure charts	Designer's perspective of the high-level structure (architecture) of the application

The third phase, architectural modeling was not applied in this feasibility study.

In this feasibility study, Kang indicates that FODA provides support for the decision-making process associated with cost assessment and performance estimates. FODA also provides a natural organization for a software reuse library. Features and functional models define the structure for organizing and populating a library and can provide insight into possible solutions to domain-specific problems.

However, Kang believes that the method is limited due to lack of representations or tools to support the method's concepts. The method is not formal, and is textual only. Issues across phases cannot be related. The benefits are theoretical, and metrics need to be collected to validate the method for future widespread use.

Kang describes FODA and the feasibility study within the general context of domain analysis. The concept of domain analysis was conceived in 1980, yet it

remains a research topic and is relatively new to the practice of software engineering. There is still no agreement on the best method, representation or the resulting products of domain analysis.

Kang establishes a set of definitions that are commonly used in domain analysis. For example, a domain is a set of current and future applications which share a set of common capabilities and data. Domain analysis is the process of identifying, collecting, organizing, and representing the relevant information in a domain. This process is based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain. The domain analysis process consists of context analysis, domain modeling, and architecture modeling. Domain analysis provides a reference model for describing a class of systems and is usually a manual activity.

A domain model is a definition of the functions, objects, data, and relationships in a domain. A context is the circumstances, situation, or environment in which a particular system exists.

Domain engineering is an encompassing discipline which includes not only the domain analysis process, but also the subsequent construction of components, methods, and tools that address the problems of system/subsystem development.

Kang believes that domain analysis helps implement reuse. The intuitive justification for domain analysis is the same for reuse, i.e., quality improvement and cost reduction. Domain analysis is a necessary first step in establishing the requirements for software reuse.

Domain analysis can provide guidance in determining what to build to support reuse and how to build it. A common model in a domain can lead to a pool of reusable resources that can be tested and measured solutions to specific sub-problems in a given application area. Reusable software developers will know what to build and how to parameterize their products for varied use across the domain, rather than overgeneralizing them for all possible contexts. Figure B-1 illustrates how domain analysis supports software development.

The maturity of an engineering field can be measured by the level of standardization of the design of products in the field. For example, in today's markets, no cars are designed from scratch; design frameworks have been standardized over time, and new features are added to an existing design framework to develop a new model. Software development, like other engineering fields, can benefit from the development and reuse of "product frameworks" in an application domain (i.e., a product line or a product family).

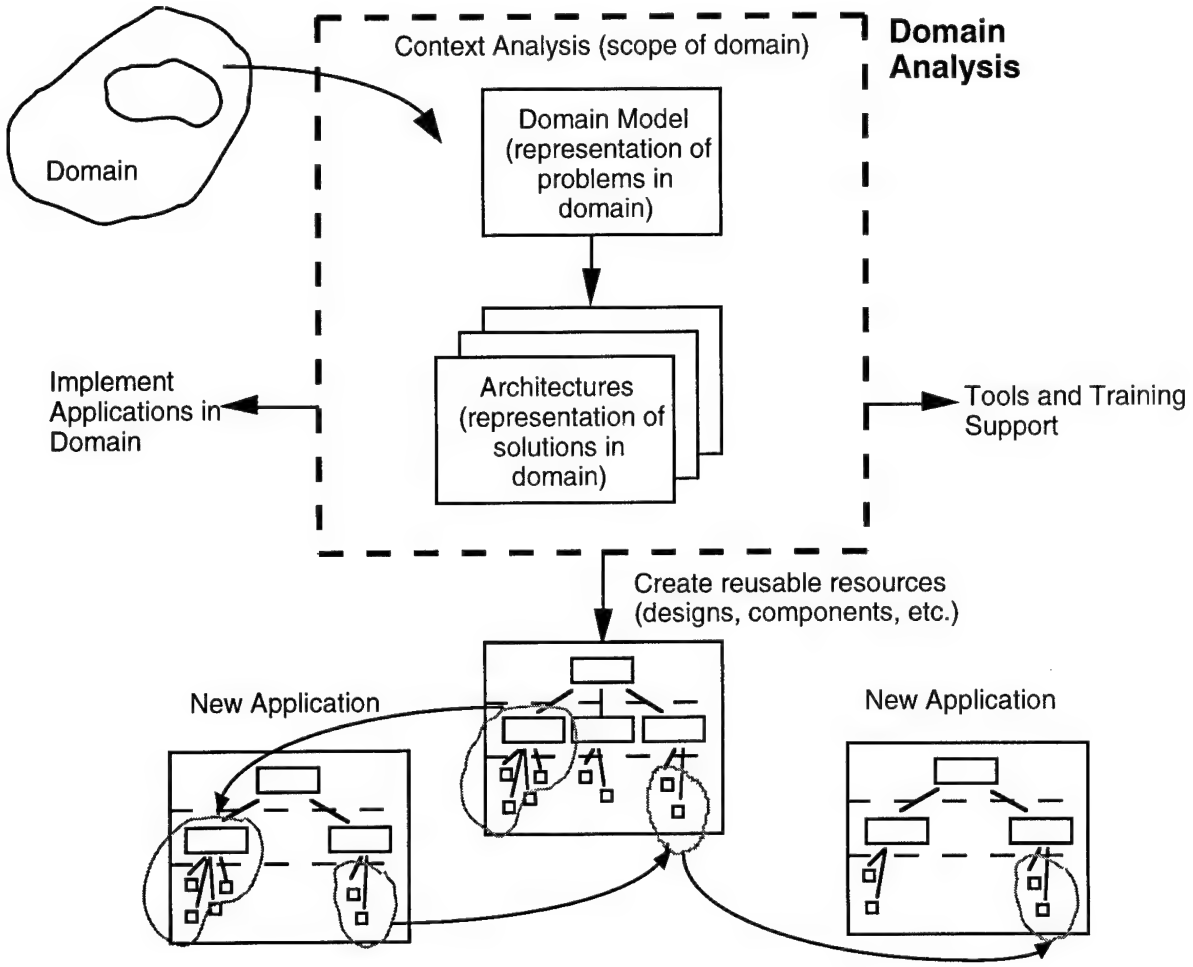


Figure B-1. Domain analysis supports software development [KAN90]

The product frameworks in the context of software are abstractions of functionalities and design (i.e., architecture) of the applications in an application domain. The modeling concept used to develop these product frameworks are aggregation/decomposition, generalization/specialization, and parameterization.

In 1990, Prieto-Diaz said that domain analysis, a systematic discovery and exploitation of commonality across related software systems, is a technical requirement for achieving successful software reuse. Domain analysis can propose a set of architectural approaches for implementing new and successful systems.

[KAT94] Katz, Susan, and Christopher Dabrowski, Kathryn Miles, Margaret Law, NIST Special Publication 500-222, "Glossary of Software Reuse Terms," Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-0001, December 1994.

Katz defines a glossary of software reuse terms in a National Institute of Standards and Technology (NIST) document and selected entries are included here to establish consistency of meaning within the reuse context.

Asset - Any product of the software life cycle that can potentially be reused. This includes the domain model, domain architecture, requirements, code, databases, database schemas, documentation, user manuals, test suites, etc.

Asset Evaluation - The process of determining whether a particular asset fits requirements and constraints of a particular software system. The definition of this standard term impacts and feeds into the area of asset selection.

Cataloguing - The process of placing information about an asset into a software reuse library. The asset plus its catalog information becomes a reusable software asset. The definition of this standard term impacts and feeds into the area of asset selection and asset classification schemas.

Domain - A distinct functional area that can be supported by a class of software systems with similar requirements and capabilities. A domain may exist before there are software systems to support it. The definition of this standard term impacts and feeds into the area of domains.

Domain Analysis - The analysis of systems within a domain to discover commonalities and differences among them. The process by which information used in developing software systems is identified, captured, and organized so that it can be reused to create new systems within a domain. The definition of this standard term impacts and feeds into the area of domains.

Domain Definition - The process of determining the scope and boundaries of a domain. The definition of this standard term impacts and feeds into the area of domain.

Domain Engineering - A reuse-based approach to defining the scope (i.e., domain definition), specifying the structure (i.e., domain architecture), and building the assets (e.g., requirements, designs, software code, documentation) for a class of systems, subsystems, or applications. Domain engineering can include domain definition, domain analysis, domain architecture and domain implementation. The definition of this standard term impacts and feeds into the area of domain.

Domain Expert - Individual who is intimately familiar with the domain and can provide detailed information to domain analysts. The definition of this standard term impacts and feeds into the area of domain.

Domain implementation - The process of creating adaptable assets that can be reused in the development of software systems within a domain. Domain implementation may also include the specification of a software development process that describes how software systems in the domain are developed through reuse of assets. The definition of this standard term impacts and feeds into the area of domain and asset production.

Domain Manager - Individual or organization responsible for managing the definition, use, evaluation, and evolution of assets within the domain. The definition of this standard term impacts and feeds into the area of domain, business strategies and asset production.

Domain Models - A product of domain analysis which provides a representation of the requirements of the domain. The domain model identifies and describes the structure of data, flow of information, functions, constraints, and controls that are included in the software systems. The domain model describes commonalities and variabilities among requirements for software systems in the domain. The definition of this standard term impacts and feeds into the area of domain.

Faceted Classification - A method derived from the field of library science which can be used to provide multiple access routes to reusable software assets in a reuse library. The definition of this standard term impacts and feeds into the area of asset selection and asset classification schema.

Horizontal Domain - A domain that provides information or services to more than one domain. Examples of horizontal domains include communications, graphical user interfaces and databases. The definition of this standard term impacts and feeds into the area of domain.

Opportunistic Reuse - The ad hoc reuse of assets in the development of software systems using a software development process that has not be altered to accommodate systematic reuse. In opportunistic reuse, the developer determines where reuse can be applied to develop a software system without the organized use of domain engineering products during successive stages of a software engineering process. The definition of this standard term is impacts and feeds into the area of business strategies.

Reusability - The degree to which an asset can be used in more than one software system, or in building other assets, with little or no adaptation. In a reuse library, reusability is the characterization of a reusable software asset that make it easy to use in different contexts. The definition of this standard term impacts and feeds into the area of reuse frameworks.

Reusable Software - Software designed and implemented for the specific purpose of being reused. Reusable software is a broad term applied to assets, applications, or software systems. The definition of this standard term impacts and feeds into the area of asset production.

Reusable Software Asset (RSA) - An asset that has been catalogued and is stored in a reuse library. The definition of this standard term impacts and feeds into the area of asset production, asset selection and reuse framework.

Reuse - to use again. The process of implementing or updating software systems using existing software assets. The definition of this standard term impacts and feeds into the area of asset production and reuse framework.

Reuse-Based Development - The use of a disciplined, systematic, quantifiable approach to the development, operation, and maintenance of software (where reuse is a primary consideration in the approach). The definition of this standard term impacts and feeds into the area of asset production.

Reuse Library - A controlled collection of reusable software assets, together with the procedures and support functions required to provide the reusable software assets for reuse. The procedure and support functions may be automated via a reuse library system. If this is the case, then the reuse library contains both the reusable software asset and the reuse library system. The definition of this standard term impacts and feeds into the area of reuse framework.

Reverse Engineering - The process of finding and reengineering an existing components so that it may potentially be reused in subsequent applications, developments, or maintenance. The definition of this standard term impacts and feeds into the area of asset production.

Salvage - The process of finding and reengineering an existing components so that it may potentially be reused in subsequent applications, developments, or maintenance. The definition of this standard term impacts and feeds into the area of asset production.

[LUB88] Lubars, Mitchell D., "Domain Analysis and Domain Engineering in IDeA," Technical Report STP-295-88, Microelectronics and Computer Technology Corporation, Austin, TX, September 1988.

Lubars describes a domain analysis process based on the information contained in the IDeA (Intelligent Design Aid) knowledge bases. The information is organized into six categories:

1. Properties - attributes describing objects in the domain, arranged in a tree structure which has the more abstract objects appearing higher in the tree. Relations between the properties are derived from the context of the domain.
2. Data types - descriptions of properties organized into a type lattice, used in classifying and selecting design schemas.
3. Design schemas - abstract solutions for a class of related design problems, arranged in a abstraction hierarchy.
4. Schema specialization rules - mappings between a design schema and a data flow design that represents a refinement or implementation.
5. Type constraints - propagate property assignments of data types to other data types that share the same abstract property class.

Information to populate these categories is derived by domain analysis.

[MYE88] **Myers, Brad, "A Taxonomy of Window Manager User Interfaces,"** ***IEEE Transactions on Computer Graphics and Applications,*** **Vol. 8, No. 5, pp. 65-84. September 1988.**

Myers developed a taxonomy for window managers and analyzed the differences in applications across domains. He showed that domain analysis can be useful for guidance in software evaluations and design.

[PER89] **Perry, James M. and Mary Shaw, "The Role of Domain Independence in Promoting Software Reuse: Architectural Analysis of Systems," Position Paper of the Reuse in Practice Workshop, Software Engineering Institute (SEI), Pittsburgh, PA, July 1989.**

Previous work-to-date has stressed the importance of application dependencies through domain analysis. Perry proposes the concept of "architectural analysis" which attempts to raise the abstraction level of design elements and thereby emphasizes domain independence. Both domain analysis and architectural analysis are related and support one another even though each has different goals and processes.

[PRI87a] **Prieto-Diaz, Ruben, "Domain Analysis for Reusability,"** ***Proceedings of the COMSAC 87: The Eleventh Annual International Computer Software and Application Conference,*** **pp. 23-29. IEEE Computer Society, Washington, D.C., October 1987.**

Prieto-Diaz proposed a domain analysis method that consists of pre-domain analysis activities, domain analysis and post domain analysis activities. Pre-domain analysis activities consist of defining and scoping the domain, identifying sources of knowledge and information about the domain, and defining an approach to the next step of domain analysis. Post-domain analysis activities include identification and implementation of reusable components and production of software reuse guidelines.

Prieto-Diaz defines the domain analysis context as a group of inputs and outputs. The inputs are domain analysis guidelines from the domain analyst, domain knowledge from the domain expert, and standard examples from existing systems. Outputs are reusable components and domain standards. Prieto-Diaz uses proven classification techniques from library science and examples from biological sciences to illustrate his concepts. This author is well-published with journal articles and books that describe his method in further detail.

[PRI91] **Prieto-Diaz, Ruben and Guillermo Arango,** ***Domain Analysis and Software System Modeling,*** **IEEE Computer Society Press, Los Alamitos, CA, 1991, ISBN 0-8186-8996-X, p. 63-69.**

Ruben Prieto-Diaz and Guillermo Arango, two well-known experts in domain analysis, published this book of selected technical papers as a tutorial on domain analysis and software system modeling. The authors discuss an overview of domain analysis concepts and research directions as follows.

Domain analysis addresses the questions of how to identify, capture, and evolve reusable information within restricted problem areas. The basic problem in domain analysis is the definition of boundaries. As defined by Neighbors, domain analysis is an attempt to identify the object, operations, and relationships between what domain experts perceived to be important about the domain. Neighbors also emphasized that the key to reusable software is captured by domain analysis in its broader focus (i.e., reusing analysis and design, not just code).

Domain analysis is a fundamental step in the creation of a reusable component. Domain analysis is performed prior to system analysis and results in a proposed a model and alternatives for automation and improvement. Consequently, the output of domain analysis is a common model across applications, with objects and their operations.

The authors believe that the most powerful sort of reuse is the reuse of analysis of information. Other areas related to domain analysis are knowledge acquisition, knowledge bases and classification schemes.

The Prieto-Diaz's method for domain analysis has three basic activities:

1. Identify objects and operations
2. Abstraction
3. Classification

Prieto-Diaz uses data flow diagrams to capture the domain analysis process. In order to perform domain analysis, the domain information is prepared, the domain is analyzed and reusable work products are produced. Out of these activities, results a requirements document, a domain taxonomy and domain frames. Modeling the thinking of this process leads to production of reusable work products using standards from the domain.

The user context of the Prieto-Diaz method is the domain analyst, the domain expert, the library and the software engineer. For Prieto-Diaz, the domain analysis process and method is formalized and experiments are needed to validate and refine it.

As for the state of the art in domain engineering, domain analysis and engineering are still undergoing states of basic research and concept formation. As of 1991, depending on industry's commitment of resources, three to five years may pass before the first domain-engineering environments consisting of methods and integrated tools and training can be expected from developing organizations and transferred to external users.

[SIM95a] Simos, Mark and Dick Creps, "ODM (Organization Domain Modeling) Guidebook Published," *STARS Newsletter* (Software Technology for Adaptable, Reliable Systems), March 1995, Issue 12, p. 11.

This newsletter articles announces the initial publication of a guidebook by the Unisys STARS team that documents how to use the domain analysis method called ODM (Organization Domain Modeling). ODM was developed by Mark Simos with sponsorship from STARS program (on behalf of the U.S. Department of Defense (DoD) Advanced Research Projects Agency (ARPA)) and the Hewlett-Packard Company (HP).

ODM has the following four features:

1. Emphasizes domain planning and domain selection
2. Focuses on comparative feature models derived from domain legacy knowledge and products
3. Has a clear distinction between descriptive models of the domain as it currently exists
4. Uses prescriptive models for the reusable assets to be developed addressing future customer needs in the domain

ODM uses an IDEF 0 process model and a set of process trees to summarize the hierarchy of the ODM process. The guidebook provides detailed process sets as inputs and outputs, each with entrance and exit criteria. The guidebook also provides example work products and templates.

ODM is at the core of the Unisys STARS reuse technology strategy. ODM is the domain engineering approach that is being developed and applied on the Army STARS Demonstration Project.

[SIM95b] Simos, Mark and Dick Creps, Carol Klingler, Larry Levine, "Organization Domain Modeling (ODM) Guidebook, Version 1.0, STARS-VC-A023/911/00 Informal Technical Report, Contract No F19628-93-C-0130, March 17, 1995.

The purpose of this guidebook is to provide a definitive ODM reference document which promotes public understanding of the method and provide practical guidance for its use and tailoring. The audience for the guidebook consists of the Program/Project Planner, the Reuse Advocate, the Process Engineer and the Domain Engineer.

The ODM is part of the Unisys STARS "Reuse Whole Product." The Reuse Whole Product includes a set of reuse support technologies that the Unisys STARS team has developed, integrated, and used. The Reuse Whole Product supports the STARS vision of mega-programming which integrates process-driven, domain-

specific reuse-based software engineering with modern tools and environments. These technologies include the following concepts:

- STARS Conceptual Framework for Reuse Processes (CFRP) - conceptual foundation and framework for understanding domain-specific reuse in terms of the processes involved.
- Reuse-Oriented Software Evolution (ROSE) process model - a CFRP-based life cycle process model that partitions software development into domain engineering, asset management and application engineering and emphasizes the role of reuse in software evolution.
- Reuse Library Framework (RFL) domain modeling toolset - a toolset which supports taxonomic domain modeling by semantic network and rule-based formalisms, features graphical and outline-based browsers.
- Capture domain modeling and legacy management toolset - a toolset which graphically supports comparative modeling of system artifacts and domain assets.
- ReEngineer - a toolset to support the reengineering of legacy systems by fine-grained analysis and abstraction of system structure.

The ODM Guidebook provides a detailed decomposition of domain engineering into process trees and IDEF 0 diagrams that define the activities and work products required for each step.

The ODM Guidebook highlights three key challenges that need to be considered when putting ODM into practice:

1. Handling anxiety concerning deferred decisions
2. Dealing with complexity and formality in the process
3. Integrating diverse skills

Decisions should be made only if it points to a project constraint, is associated with a risk or uncertainty, or a long lead-time is required. Adopting formality incrementally, or only as required, may help cope with the burden of ODM's formal process. Team modeling and training may help to integrate diverse skills, as does treating domain engineering as a personal and professional discipline.

The ODM has been applied on a small scale by a variety of organizations, including Unisys, CARDS, and the SEI. Major ODM applications at Hewlett-Packard and the Army STARS Demonstration Project have produced good results. The Reuse Whole Product effort will continue through early 1996.

[SPC91a] Software Productivity Consortium (SPC), "Domain Analysis Workshop Presentations," SPC-91186-MC Version 01.00.00, September 26-27, 1991, Herndon, VA.

The workshop was a forum for exchanging of ideas in the current research area of domain analysis. The workshop was attended by representatives from the following companies: Harris Corporation, Rockwell, Hughes, United Technologies and General Dynamics. Presenters began by introducing a definition of domain analysis published by Jim Neighbors in 1980; domain analysis is the activity of identifying objects and operations of a class of similar systems in a particular problem domain.

The current practice of domain analysis is ad hoc. A practitioner gains experience by constructing several of the "same kind" systems, using experience to identify and isolate recurring operations for encapsulation and standardization.

Mitch Lubar briefed the attendees on his use of Reuse-Oriented Software Evolution (ROSE), a tool to support domain analysis. Lubar defines domain analysis as the analysis of an application domain that leads to the construction of a domain model for the purpose of solving problems in the domain. Domain analysis can be considered as an activity that analyzes an application domain for reusability. Domain analysis requires expertise in the application domain; that is, knowing what to look for and knowing how to interpret and abstract the commonalities and differences. Domain analysis is time consuming and tedious and requires mature and stable domains.

Sidney Bailin presented his experience with KAPTUR, a tool to support domain analysis. Bailin believes that bounding the domain is the key to solving the analysis problem. Too broad of a domain can lead to a superficial model which is not very useful. He provided examples of domains and discussed how he established their bounds.

Grady Campbell, Jr. presented a case study to substantiate his belief that the purpose of domain analysis is to characterize the problems, solutions and production processes appropriate to a domain.

Neil Burkhard briefed the attendees on his case study of domain analysis using the ATD/CWM data set.

Kyo Kang presented a feasibility study using the FODA method of domain analysis. Kang believes that domain analysis is the systematic exploration of related software systems to discover and exploit commonality. He defines a feature as a prominent or distinctive user-visible aspect, quality, or characteristics of a software system or systems. Attributes of a feature are rationales, composition rules, issues and binding time.

Ruben Prieto-Diaz presented a domain analysis process model. His model is procedural and based on a methodology for deriving specialized classification schemes. He feels that domain analysis is similar to deriving faceted classification

schemes since both are aimed at finding generic characterizations and standard models. Prieto-Diaz defined the following steps in a domain analysis process:

1. Select representative samples from a collection of titles.
2. Identify common terms.
3. Abstract, classify and give structure to make a model.
4. Use the model as a classification standard.
5. Update structure as the collection grows.

Prieto-Diaz presented examples of faceted classification using Booch components and components from Command and Control Systems.

The workshop concluded with presentations of case studies of domain analysis by Dan Benson of the ADGE System Architecture for the Air Defense Ground Environment and by Patty Franck of the Avionics Display domain.

[SPC91b] Software Productivity Consortium, "Synthesis Workshop," September 23-25, 1991. Herndon, Virginia.

Synthesis, a domain analysis method developed by the Software Productivity Consortium (SPC), uses the paradigm of flexible production lines as a model for the process of creating and developing software. In other engineering disciplines, the production process has evolved from handcrafting, to repeatable engineering, to production lines, and finally, to flexible production lines that leverage commonality across variations. To establish and evolve a production line, a product family is defined; concurrent engineering of product and process is performed; the needs of the customer are matched to the product and process; and adjustments are made for optimization.

Because software development is an intellectually, non-deterministic process and the product is invisible, complex, non-uniform and changes repeatedly, F. Brooks claims there are "No-Silver Bullets" to success. However, by combining an analysis of a particular domain and defined business objectives, an organization can define a product family and flexible production lines to produce successful family members that can be produced and modified in response to customer feedback.

By reorganizing the development process to follow a production line model, an evolving family of products is created, requirements refinements are separated from implementation, and the requirements refinement process is optimized to the organizational and customer needs. This evolving family of products positions an organization to create a flexible production line and defines a decision model, a standard engineering decision process, and mechanically adaptable design components. By linking decisions to components, adaptation and integration flexibility in products and their production is provided.

Synthesis is an approach for developing similar software applications based on systematic reuse and knowledge and products. It is based upon families of systems and components, abstraction and adaptation. Abstraction leverages commonalities across families of similar systems or across the life cycle of a large single system. Adaptation distinguishes decisions needed to identify a particular member of the family. Figure B-2 illustrates the Synthesis process.

The Synthesis method of domain analysis results in a high level model of a system and systematic reuse of domain and software engineering knowledge and products. In addition, Synthesis supports a cycle of rapid requirements refinements. Synthesis combines the disciplines of domain engineering and application engineering (i.e., generating a product in a budgeted time for a pre-defined cost).

Synthesis benefits customers and the development organization. Customers profit from rapid handling of changes to requirements, better matching of the product to their needs, improved communication, reduced cost and schedule and consistent product quality. The organization profits by capture and leverage of product area expertise and software design knowledge and improved risk management. The principles of Synthesis are families of systems, model-based specification and analysis, large scale, systematic reuse and mechanical product generation.

Synthesis can also be seen as an approach for systematic and effective reuse. Synthesis is a way to put assets to work, (i.e., technical knowledge, business knowledge, engineering experience and problem solutions). Synthesis addresses requirements, verification, integrated methods and toolset, prediction and measurement, as well as reuse. The process to create a domain definition results in a domain synopsis, a glossary, assumptions, viability analysis and domain status.

The Synthesis Workshop demonstrated the Synthesis method as applied to the Host-At-Sea (HAS) Buoy System. Lessons learned when applying the Synthesis method to realistic examples are the following:

- Disagreement on bounds and interpretation of domain was prevalent.
- It was difficult to determine when activities were completed.
- Variations and iterations were unclear.
- Heuristics were needed.

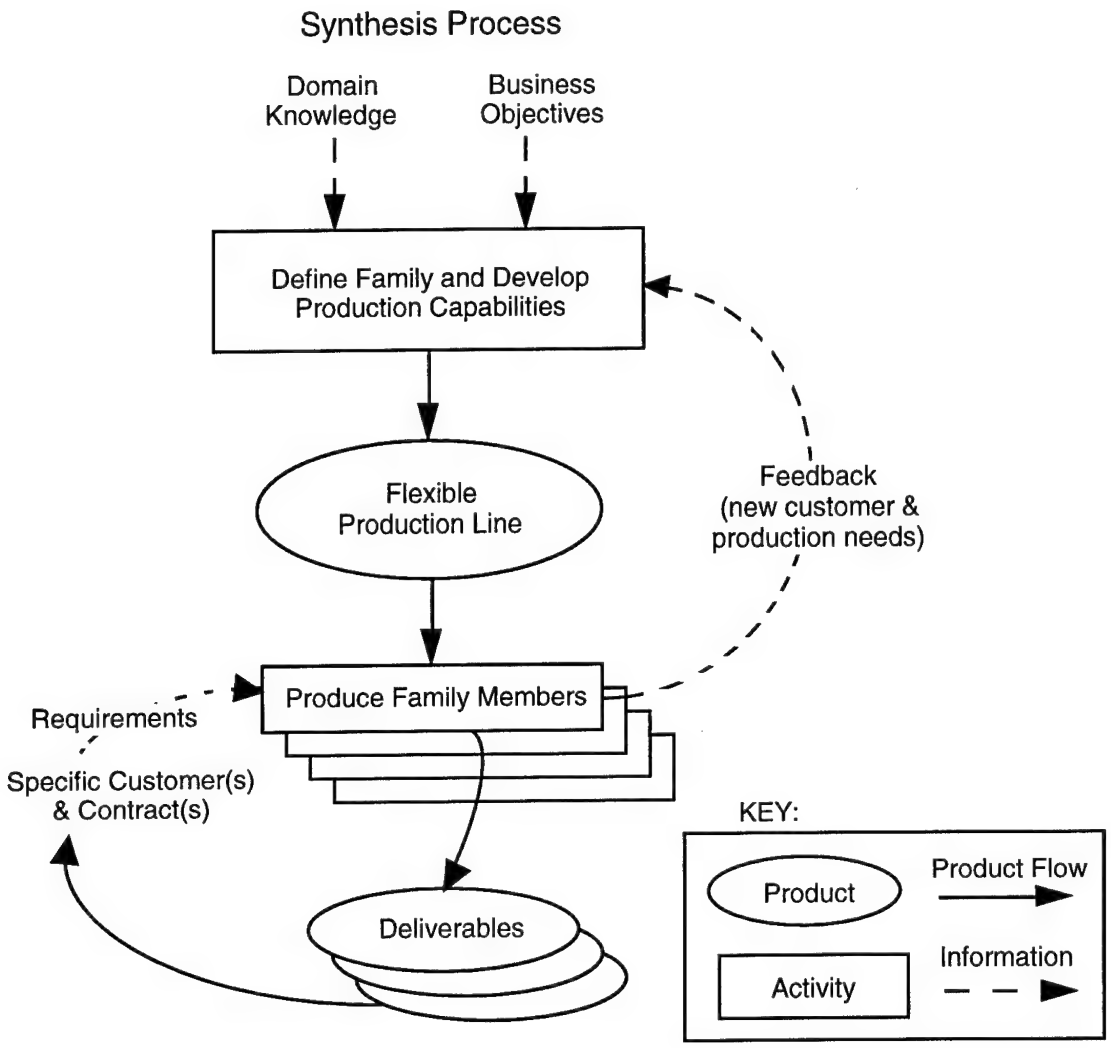


Figure B-2. The Synthesis process [SPC91b]

Produced at a later date through the work at SPC and the Synthesis Workshop, the Synthesis Guidebook [SPC92a] defines the method and the Case Studies [SPC92b] documents an initial validation of the method. In 1991, SPC had plans to evolve existing application generator tools to support the Synthesis method.

[SPC92a] Software Productivity Consortium, "Synthesis Guidebook, Volume 1: Methodology Definition." SPC-92111-CMC, Version 01.00.00 October 1992.

Synthesis is a methodology for constructing software systems as instances of a family of systems that have similar descriptions. Synthesis encompasses application engineering and domain engineering. Application engineering is how a group (or project) in an organization creates a product to meet customer requirements.

Domain engineering is how an organization improves productivity by creating an application engineering process, tailored for a project in a particular business area and supporting standardized, reusable products.

The context for the Synthesis methodology is determined by three concerns:

1. Business objectives
2. System engineering practices
3. Software engineering processes

The first consideration of the Synthesis method is to establish business objectives, working in conjunction with an organization's management philosophy. Business objectives should consider the expertise in the organization, the customers' future needs and changing technology. Business objectives are based on a family of systems and understanding their similarities, leveraging production of high quality, providing reliable systems at a lower cost. An organization's business areas and product lines help to determine its business objectives. Business objectives help to define the domain of an organization.

Another consideration in the Synthesis method is system engineering practices, specifically, partitioning a system problem into manageable subsystems. System engineering maintains a "big picture" view, and as such, is needed for domain engineering and application engineering within Synthesis.

The last consideration in the Synthesis method is software engineering processes. Software engineering processes are derived by analysis, synthesis, evaluation and management. The Synthesis method supports family-oriented software development and abstraction-based reuse which is a part of software engineering processes.

The Synthesis Methodology Reference Model was developed by Campbell in 1990 as a canonical definition of the processes, products and activities of Synthesis. The Synthesis Guidebook defines the following hierarchy of domain engineering activities within the Synthesis methodology:

- Domain Engineering
  - Domain Management
  - Domain Analysis
    - Domain Definition
    - Domain Specification
      - Domain Model
      - Product Requirements
      - Process Requirements
      - Product Design
        - Product Architecture
        - Component Design
        - Generation Design
      - Domain Verification
    - Domain Implementation
      - Product Implementation
        - Component Implementation
        - Generation Implementation
      - Process Support Development
    - Project Support

Synthesis and these activities are based upon the following key principles: Program families, iterative processes, specification and abstraction-based reuse. Abstraction of similarities support a form of standardization that enables systematic adaptation to meet the specific needs of a particular customer. Synthesis also identifies decisions that must be deferred until a particular system is needed. Synthesis parameterizes a work product to show how it varies as a result of those decisions; consequently, the work product is made adaptable.

One major advantage of Synthesis lies in its emphasis on long term objectives for business and technology development. Synthesis exploits similarities, eliminates redundant work, focuses on resolving variations to satisfy needs of an organization and its customers. Synthesis focuses on quality products that are profitable.

Along with the guidebook documents, SPC requires training to use the Synthesis method and to apply the method to pilot projects. The current guidebook applies to exploratory stages of software development and reuse. In the future, SPC plans to develop additional guidebooks for the developmental, functional and production stages of software development and reuse.

[SPC92b] Software Productivity Consortium, "Synthesis Guidebook, Volume 2: Case Studies," SPC-92111-CMC, Version 01.00.00 October 1992.

This second volume of a two part series documents a case study of applying the Synthesis method to the ATD/CWM (Air Traffic Display/Collision Warning Monitor) from the ADARTS program. The guidebook was developed from the Synthesis Workshop [SPC92a] and the Synthesis method of domain analysis was applied to the communications and control and management systems domain at Rockwell International Communication and Control (RICC).

Appendix C - Annotated Bibliography of Asset Production

The information gleaned from this literature survey of asset production was used to determine the operational context of the Reuse Context for Asset Quality Certification and to assess the impact of this previous research on the development of the Certification Framework.

The annotations in this appendix summarize the essence of each of the referenced publications. Summaries vary in length; those that are longer provide additional details because the reference appeared to be a flagship among others. The shorter annotations were still included to serve as a pointer to the complete reference if more details are of interest.

This annotated bibliography this area is not exhaustive, but gives a flavor of the previous research that has been accomplished. Some of these references were used in other appendices.

[ADE85] Adelson, Beth and Elliot Soloway, "The Role of Domain Experience in Software Design, *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 11, November 1985, pp. 1351-1360.

Adelson showed in her case study that designers with inadequate domain knowledge were quick to constrain their designs to secure sufficiently specific models for their simulations. Designers with previous experience in their assigned domain used existing plans rather than formulating new ones. The techniques of note-taking and simulation were used when the designers had prior knowledge of the domain.

[ARA89] Arango, Guillermo F., "Domain Analysis - From Art to Engineering Discipline," *Proceedings of the Fifth International Workshop on Software Specification and Design,* IEEE Computer Society, Washington, D.C., May 1989, pp. 152-159.

Arango feels that within the reuse community, there is a well-founded belief that domain analysis will facilitate the identification and capture of reusable abstractions for restricted classes of applications. These reusable abstractions aid in the production of reusable assets.

[BAI88] Bailin, Sidney, "Semi-Automatic Development of Payload Operations Control Center Software," NASA Goddard Space Flight Center, Computer Technology Associates, Laurel, MD, October 1988.

Bailin applied domain analysis to the software of a Payload Operations Control Center (POCC), and proposed an approach for semi-automatic development of software for an application processor software based upon his results. In his example, he abstracted typical components and identified patterns for commonalities and differences to facilitate reuse. He used a mix of constructive and generative technologies as driven by the different parts of the application processor.

[BAI89] Bailin, Sidney, "Generic POCC Architectures," NASA Goddard Space Flight Center, Computer Technology Associates, Laurel, MD, April 1989.

Bailin developed a generic architecture for the POCC based on abstraction and object-oriented design principles to form the basis of engineering a rapid synthesis environment. The generic architecture can be used as a specification for assets and their production.

[BAN93] Banker, Rajiv D., Robert J. Kauffman, Dani Zweig, "Repository Evaluation of Software Reuse," *IEEE Transactions on Software Engineering*, Vol. 19, No. 4, April 1993.

Banker evaluated repositories and their impact on software reuse and provided the following insights. Some application domains are more conducive to reuse. The success of guiding a user through the repository's cataloguing scheme determines the reuse opportunities that are visible to the user. The reuse of specialized components are constrained by adaptation costs.

[BAT88] Batory, Don S., J.R. Barnett, J. Roy, B.C. Twichell and Jorge F. Garza, "Construction of File Management Systems for Software Components," Technical Report TR-88-36, University of Texas, Austin, TX, October 1988.

Batory used domain analysis of existing file management systems to discern a generic architecture. Based upon published algorithms and accepted data structures, his generic architecture served as a template into which building block components can be plugged. The architecture facilitated standard interfaces making components interchangeable. His in-depth study showed the assembly of simple systems from pre-written components, demonstrating savings in cost and schedule.

[BIE95] Bieman, James M. and Santhi Karunanithi, "Measurement of Language-Support Reuse in Object-Oriented and Object-Based Software," *Journal of Systems Software*, 1995: 30: pp. 217-293.

Bieman points out that measures and measurement tools to quantify language-supported reuse have been lacking. Even though emerging object-oriented software development techniques and languages have the potential for improved reuse, little work has been to validate the benefits. Reuse measurement will help users gain insights to develop software that is easily reused.

Reuse is not simple to classify and measure. Many perspectives can be used. Reuse can be measured from the perspective of the client, server, or from the system perspective. Many of the attributes can be quantified with simple counts such as the number of direct or indirect servers, the number of server class instances, the number of object instances, the number of calls to a method in the server, the number of library units that are visible to a unit, and the number of library units imported explicitly in a unit.

Empirical results can help determine the amount of reuse in existing systems and identify the most frequently reused software components. Tools that measure attributes related to reuse in software systems can identify properties that make software more reusable. Bieman developed a prototype tool called Ada Reuse Measurement Analyzer (ARMA). He performed an initial empirical evaluation of

the tool using data from a commercial software system. He provides the following insights:

1. Having a single, visible data type per package increases the reusability of that package.
2. Too many levels of nesting of units lowers reusability.

Multiple nesting levels requires developers to be knowledgeable of all levels in order to use the nested unit.

[BIG87] Biggerstaff, Ted and Charles Richter, "Reusability Framework Assessment and Directions," *IEEE Software*, March 1987.

Biggerstaff notes limited success stories with reuse of code, but points out that numerical computation routines are highly reusable. This difference is due to the characteristics of this domain. The numerical computation domain is unique in several ways:

1. The domain is very narrow and contains only a small number of data types.
2. The domain is well-understood since the mathematical framework has evolved over hundreds of years.
3. The underlying technology is quite static, growing and evolving very slowly. It evolves so existing parts of the technology remain unchanged with upward compatibility within the technology.

All three of these characteristics lead to establishment of standards and components that have a high probability of reuse.

A mature domain positively affects understanding of the problem domain and reduces the long term investment in a reuse library. Narrowness of a domain makes reuse manageable, and the cost of developing parts is small since only a few data types exist. A library can consist of stable parts and the investment can be amortized over a long period of time. Unlike the numerical computation domain, the worst kind of domain for reusability is one where the underlying technology is rapidly changing.

[BIG89] Biggerstaff and Alan J. Perlis, "*Software Reusability, Volume I, Concepts and Models* and *Volume II, Applications and Experience*," ACM Press New York, NY, 1989.

Biggerstaff coins the acronym of VLSR (Very Large Scale Reuse) as the best course of action to realize the full potential of reuse. His rationale for this expansive view of reuse is that the more narrowly defined views of reuse have not shown a very large return on investment (i.e., replication of code, reuse of subroutine or

object libraries, reuse of Ada packages). Usually these types of reusable components have a high degree of specificity and tend to be small in size. Building systems out of these small components requires designing the architecture that binds the components into the whole system. The cost to build this superstructure is typically much larger than the savings of reusing a set of small components.

Making components larger to offset this problem, produces yet another problem. As code components become larger and larger, they are less likely to be used. The specialized nature of these components reduces the opportunity that they will be reused. Very large components also require a significant effort to understand and adapt to a new system. Biggerstaff contends that code-oriented reuse should be a standard operating procedure, yet reuser should be ever cognizant that this type of reuse has limited gains. It is the VLSR that holds the full potential of maximum benefits of reuse.

Using VLSR, the representation of a component is sufficiently general to allow reuse over a broad range of target systems. Representations should allow a large-grain component structure to be described precisely while leaving many of the small, relatively unimportant details uncommitted. Representations should allow a broader range of information to be specified than source code (i.e., design structures, domain knowledge, design decisions, etc.).

Biggerstaff strongly believes that the greatest potential payoff is in representational breakthroughs that solve the problems of factored forms, partial specification, the coupling of instances and their interpretations, and controlled degrees of abstraction. He believes that the key to solving these types of problems is the notion of semantic binding, or binding by analogy. This form of binding, applying a design from one context to a new and different context, will provide the most general method for reuse.

Biggerstaff also advises those creating libraries of reusable code components. Once assets are produced, the storage library should be based on a standard for the domain-specific types of the data consumed and produced by the components in that library. If none exists, then the level of reuse is likely to be low. He discourages "finding" and "throwing together" a bunch of components that have functions that more or less cover the needs of the using organization. A library of components needs to be designed according to a common architectural guidelines that reflect both the nature of the problem domain as well as the computational needs of the organization.

Biggerstaff agrees with Parnas' early design organizing principles, that is, information hiding or encapsulation. This principle enhances the reusability of components because of the isolating effect of information hiding; it allows the components to be reuse in a black box mode. Even if modifications must be made, they are easier to make because all of the information pertaining to a specific module is hidden or organized within the module rather than being randomly scattered about the overall design.

Biggerstaff believes Parnas' work is significant in that it applied a theory to a large-scale problem, an accomplishment that few other researchers can claim. Parnas and his colleagues have spent nearly ten years redesigning the avionics software for the A-7E fighter aircraft according to the principles of information hiding.

[CAL91] Caldiera, Gianluigi and Victor R. Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer*, February 1991, pp. 61-70.

Caldiera maintains that software production using reusable components will probably be crucial to the software industry's evolution to higher levels of maturity. Caldiera points out that the development experience along with the software objects produced holds the most value for cost-effective and efficient reuse.

One of the difficulties in reusing software arises from the nature of the objects to be reused. With software, it is difficult to separate the object apart from its context. Programs and parts of programs, specifications, requirements, architectures, designs, test plans, test cases are all related to each other. The reuse of each software object implies the concurrent reuse of other objects and associated information. More than just the code is reused. All these objects have a history and may carry a large amount of expertise. It is this experience that is critical to reuse software of objects.

Another difficulty in reuse is the lack of a set of reusable components, despite the large amount of software that already exists in the files of software developers. Reuse efficiency and cost effectiveness require a large catalog of available, yet useful, reusable objects. Attempts to construct reuse libraries have fallen short of the mark.

Caldiera's model for reusing software components splits the traditional life cycle into two parts. One part, the project, delivers software systems, while the other part, the factory, supplies reusable software objects to the project. If reuse does occur, it usually is at the project development level, where reuse is difficult because a project's focus is the delivery of the system. Packaging reusable experience is a secondary concern, if at all. Moreover, project personnel cannot recognize the pieces of experience appropriate for other projects to reuse. Traditionally, existing processes of development do not include both these aspects of reuse; reuse is usually an informal sharing of techniques and project among people working on the same or similar projects.

Using Caldiera's model, the component factory supports the project development with the object and its packaged experience. It is the component factory that develops and packages software components rather than the project. It supplies code components to the project upon demand, creates, and maintains a repository of components for future use. The component factory understands the project context and can deliver components that fit since the component factory gathers the experience-base from the project.

When software project engineers have identified the system components, usually after preliminary design, they request components from the component factory and integrate them into the project. The project engineers may also request a list of components from the component factory that satisfy their given specification.

When the component factory receives a request from the software developers on a project, it searches its catalog of components to find a software component that satisfies that request, with or without tailoring. If no component approximates the request, or if modification of an existing component is too costly, the component factory develops the requested component from scratch or generates it from more elementary components.

To produce software components without specific requests from the project organization, the component factory needs to develop a component production plan. The plan can be constructed from the extractions of reusable components from existing systems or from generalizations from previously produced components by the software developers. A typical component production plan would contain common data structures and the main operations on them, implemented in desirable languages.

A component factory can develop an application-oriented component production plan by analyzing an application domain to identify the most commonly used functions. Then, it can implement these functions into reusable components to be used by the software developers. On the other hand, the factory can generalize a pre-existing components into new ones by adding more functionality or parameterizing it. Caldiera believes that this is the best model to follow for successful asset production.

[CAR87] Carle, Rick, "Reusable Software Components for Missile Applications," *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse,* Syracuse University and University of Maryland, Blue Mountain Lake, NY, July 1987.

Using domain analysis, Carle developed and modeled the requirements of a software composition system based on reusable components in his case study of the Raytheon Missile Systems Division. Then, a reusable software library was seeded with components that complied with his generic requirements. He built library access tools to demonstrate his concept.

[CHA91] Chidamber, Shyam R. and Chris F. Kemerer, "Towards a Metrics Suite for Object-Oriented Design," OOPSLA '91, pp. 197-211.

Metrics designed for object-oriented technology are emerging and may provide indicators of whether or not a software code module is produced with a goal of reusability. For example, CBO (Coupling Between Objects) and WMC (Weighted

Methods Per Class) are candidate metrics that be can be used to imply the reusability of object-oriented source code. CBO for a class is a count of the number of non-inheritance related couples with other classes. CBO indicates how much data within a class is available to other objects. In order to improve modularity and promote encapsulation, inter-object couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design; consequently, reuse and maintenance are more difficult. WMC is the number of methods implemented within a class (not all methods are accessible within the class hierarchy). Classes with many methods are usually specific to one application and may be difficult to reuse in other applications. Both these object-oriented metrics show promise to determine the reusability of produced code. Using these software engineering principles of loose coupling and generic methods for classes enables design for reuse.

[DIS95] Defense Information Systems Agency Center for Software, DoD Software Reuse Initiative, "Software Reuse Business Model (SRBM)," Technical Report, January 31, 1995.

Asset production is a significant part of the Software Reuse Business Model (SRBM) discussed in [DIS95], Appendix B, Business Strategies. In the SRBM, the following activities of Domain Management lead asset production:

1. Plan for asset production.
    a) Identify programs, end products, mission needs, etc. that the assets will be developed to support.
    b) Identify and resolve programmatic issues related to development, use and transfer of assets
        I) Identify need dates, integrate development schedules, identify the nature of the work product that will be transferred.
2. Specify asset development process standards.
    a) Identify process criteria that a mature practice of asset development should meet for domain management, analysis and implementation.
3. Specify domain/asset requirements by identifying commonalities and variabilities in needs and/or functional requirements.
4. Specify domain architecture and asset interfaces by identifying standard architectural designs for systems in the domain.
    a) Include variabilities in the design to meet variability in the domain requirements.
5. Categorize assets by identifying support for assets.

a) Categorize according to structure identified in the domain model (i.e., domain requirements, architecture or a generic schema for a domain-independent library).

6. Specify asset usage support.
   a) Include tools and process guidance.
7. Implement/buy assets by identifying and applying quality and support criteria for assets submitted to the library.
   a) Develop, procure, license, etc. the assets.
8. Maintain assets.
   a) Include upgrades and modifications.

By using this process for domain management, the Software Reuse Business Model uses both domain analysis and business strategies to drive the production of reusable assets.

[DOD95a] Department of Defense, "Software Reuse Symposium," March 23, 1995, Huntsville, Alabama.

GenVoca, an architecture specification and instantiation method, was developed by Don Batory at University of Texas, Austin, TX, and presented at the Software Reuse Symposium in 1995. GenVoca consists of an architecture representation in the form of a language specification. For each architectural building block, Batory defines a set of interface specifications for that realm or component. For each implementation of the component, he defines a set of design rules that controls the integration of components.

GenVoca has been used in several different domains; in the Intelligence-Electronic Warfare (IEW) domain on the Army Demonstration Project; in the avionics domain at Loral/DSSA, and in the domains of databases, data structures and network protocols in academic research projects.

EDGE (ELPA[1] Domain Generation Environment), is a tool developed by Unisys working for the Army in support of the STARS program. EDGE provides automated support for the development of architectures and assets. EDGE supports the GenVoca architecture specification method within an Ada context. It uses a component composer/system generator and produces Ada packages in GenVoca's layout editor.

---

[1] ELPA is an acronym for the Emitter Location Processing & Analysis system, a subdomain of the IEW (Intelligence-Electronic Warfare).

[FAC94] Facemire, J. Jeff , Aleisa Petracia, and Stephen Riesbech, "Software Architecture Seminar Report," Software Technology for Adaptable Reliable Systems (STARS), Central Archive for Reusable Defense Software (CARDS), Informal Technical Report, Contract No F19628-93-C-0130, January 29, 1994.

Jeff Facemire reports that the goals of this STARS-sponsored seminar and workshop were to understand the various meanings of software architectures, current research in the field of architectures, and current efforts in applying software architectures. STARS is directed by one of the key components of the DoD Vision & Strategy, that is, to develop architecture-centric reuse by defining reusable-oriented flexible architectures for DoD domains. These domains should be well-supported by industry and the R&D community. STARS members feel that this emphasis would spur the investment in generic software components and tooling as well as facilitate developing systems that comply with approved architectures.

The IEEE Standard Glossary of Software Engineering Terminology defines an architecture as the organizational structure of a system or asset. An architecture can include the structure of components, their interrelationships, and principles and guidelines governing their design and evolution over time. It is recommended that more specific terms be used when describing architectures. Examples of specific terms are domain architecture, software system design, strategic architecture, enterprise architecture, standards architecture, logical and physical architecture, and hardware and software architecture. Each term has unique characteristics and may have unique applications. Descriptions of architectures should include issues of standards, procurement, business, and reuse.

Even though the terms may not be used, the concept of architectures and reuse are commonly practiced in more mature disciplines such as chemical engineering. Handbooks, published processes or architectures are available within this domain, and many chemical engineering corporations have standard designs. Chemical engineering is field based upon empirical observations, scientific theory and economics. Table C-1 characterizes the differences between chemical engineering and software engineering and points to potential software engineering areas that need maturation.

Table C-1. A comparison of chemical engineering and software engineering

Chemical Engineering	Software Engineering
One main handbook for the entire field	Fragmented set of handbooks
Comprehensive coverage of unit operations	Incomplete coverage of component/algorithms
Patterns of unit operations	Few patterns
Numerous heuristics	Some heuristics
Over 100 authors	One or few authors
Emphasis on economics	Emphasis on processing and memory
Common language within chemistry	Proliferation of languages and design notations (i.e., Ada, C, C++, Booch)

An architecture can serve as a common reference point, or a way to communicate the elements of a system. Architectures help bound the problem by defining the problem space and therefore, the solution space. Development of architectures relies upon creation of a domain model. Architectures assist users to pick out constraints and create specific applications. With architectures, fatal combinations of components within an architecture can be identified prior to implementation. The following guidelines can be used to define an architecture:

1. Describe the basic elements that make up the architecture.
2. Define the rules for how the elements interact with each.
3. Describe how these basic elements make up the system design and operate within its context.

Architectures are a framework, a behavior description, and the basis for extension and customization. Currently, reuse is a scavenging process, or a parts-oriented approach. Reuse is really about generalization, layering, connectivity, collective behavior, and non-point solutions. Architectures deal with generality and its costs, modularity and it costs, shifting complexity by layering (abstraction) and generalization.

Architectures must also address the non-functional requirements of a system (e.g., interoperability, ability to tolerate change, cost to build, use of COTS). A requirement of openness gives rise to issues of compatibility and interoperation among differing standards. Resolution of architectural design issues can be demonstrated through a prototype.

In very general terms, architectures consist of elements, form and rationale. Architectures can have several common characteristics; for example, identifiable design elements, patterns, style, context and adaptable form, physical ties, and ontological structuring. The following categories of architectures were identified:

1. Data flow systems - batch, sequential, pipes and filters

2. Call and return systems - main program and subroutines, object-oriented systems, hierarchical layers
3. Independent components - communicating processes, event systems
4. Data-centered systems - transactional database, blackboard of shared systems, representation and opportunistic execution

Some work has been done to date with developing domain-specific software architectures (DSSA) to deal with a set of related problems, but not equivalent solutions. DSSA is a bottom-up approach, whereas common architectures need to span across applications. While architectures have been in the software development community for some time, the current emphasis is on their formalism. With formalism, each separate piece of an architecture becomes better defined and standards can begin to emerge.

Why is developing generic software architectures and their formalisms a difficult problem? One obstacle is that many diverse applications and languages exist (e.g., real-time, information systems, Ada, C, C++, Assembly). Each system is unique and lacks overriding standards. Diverse design approaches abound and structured design or object-oriented design is needed provide abstraction. Reuse and software using large-scale existing components (e.g., architectures) promises to significantly reduce development costs; however, the savings have been historically difficult to achieve.

Commonality among solutions is also difficult since software companies have different business goals. Establishing generic software architectures is confounded by the fact that software engineering has very few guiding engineering principles as compared with other more mature disciplines. Achieving generic architectures requires a shift in the thinking paradigm for possible solutions. The following technologies have emerged as applicable to development of software architectures:

1. Application composition (i.e., formalism, infrastructure)
2. Techniques for reusable components
3. Legacy system/software (i.e., extraction, reuse in current form)

Supporting these emerging technologies are those that are considered "low hanging fruit," that is, easily attained and useful. Low hanging fruit have been identified as object-oriented development and re-engineering, formalisms for composition, interconnection techniques, programming with parameters, consensus definition of architecture, inductive analysis of current examples, and Very High Level Definition Languages (VHDLs).

Why do we need software architectures? As shown in Figure C-1, many factors in today's industry point to the need to reuse and how generic architectures might provide a mechanism for reuse.

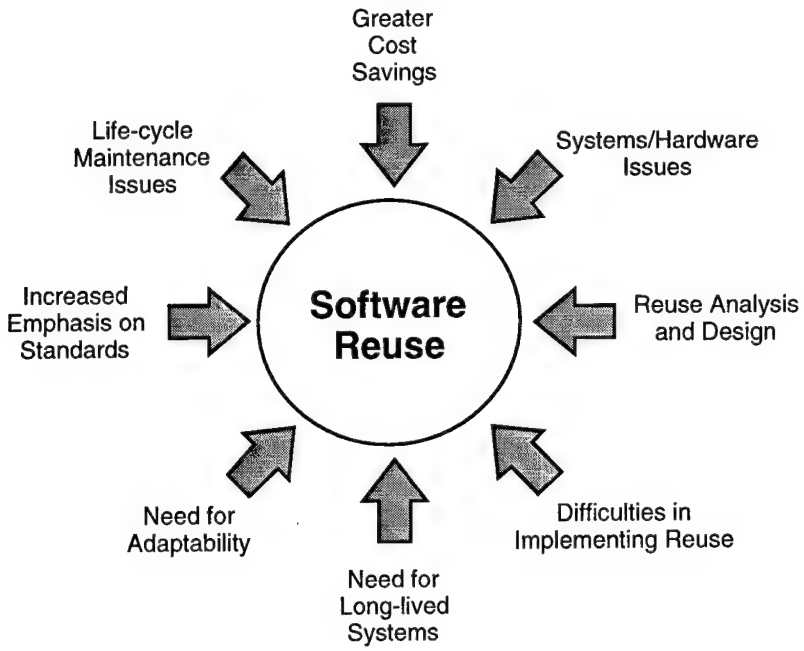


Figure C-1. Factors pointing toward reuse

A significant relationship exists between architecture and software reuse. High level analyses and designs are accompanied by context information and this information can be reused. Architectures provide a partitioning strategy and abstraction mechanisms. The higher level at which artifacts are reused, the greater the payoff. Since architectures should include a high level description of data and process views, they are optimal for reuse. A small domain is more vulnerable to external architectural constraints while a large domain is fed by a large number of resources. A generic architecture may require trade-offs since most systems are specialized and designed for optimization.

Generic architectures can promote reuse, but similar to the concept of software process improvement, this type of reuse may require a change in the way an organization does its business. Software and its architecture must be understood and become an item of capital investment that is managed. Many making these decisions have little software background to understand its problems and issues. Organizations need to manage architectures as part of their business process.

The Government and the acquisition process can make use of these generic architectures. Architectural models can be specified in Statements of Work (SOWs) as long as a specific product is not specified. The Government cannot specify a single system, only its requirements. The contractor says how they will fulfill the requirements in their architectural solution and their ultimate product.

Pioneers in the architecture-based reuse tools were DRACO and ROSE-2 (Reuse Oriented Software Evolution Model). Currently, LaSSIE, Kapture, UNSA and

Technology Book are under development. Emerging architecture-based reuse tools are LILEANNA and μRapide both featuring integrated tools and libraries. Facemire defines the "money test" as "if it doesn't attract investment beyond a single project or system, it isn't an architecture for reuse."

The Association for Information and Image Management (AIIM) states that an architecture with a mixture of object-oriented and event system characteristics is best suited for supporting reuse of architectural design and code. The rationale supporting this claim is that object-oriented design of classes and methods spawn events which are loosely coupled. Current trends are toward intersection of object orientation and event systems. Consequently, the maximum reuse potential appears to exist within the CORBA (Common Object Request Broker Architecture) and its intersection among object-oriented design and event-based transactions. The disadvantage of event systems is that indirection overhead may be high, special purpose languages may be limited, components have loose control. A component does not know who is responding to each event making it difficult to reason about correctness.

The creation of generic architectural components may be independent of the development of fielded production systems. STARS feels that developing reuse processes and standards can facilitate development of reuse conventions. Using the command center domain, the goal of CARDS is to transfer domain-specific software reuse into mainstream DoD procurements. Another demonstration project may be emerging for real-time systems, but it is still immature since the domain definitions are currently problematic.

[FOW95] Fowler, Glenn S., David G. Korn and Kiem-Phong Vo, "Principles for Writing Reusable Libraries," ***Proceedings of the Symposium on Software Reusability , SSR'95, "*** **Seattle, WA, April 28-30, 1995.**

Fowler writes from his experience over the last ten years with the reuse program at the Software Engineering Research Department at AT&T. The primary goals in building reusable components are applicability, efficiency, ease of use, and ease of maintenance. However, there is no simple set of rules that guarantees the simultaneous achievement of these goals. Often the goals conflict and decisions have to be made to trade off constraints. As a result of his work, Fowler deemed following design characteristics as important and should be used as guidelines in producing assets:

- Necessity
- Generality
- Variability
- Efficiency
- Robustness

- Modularity
- Minimality
- Portability
- Evolvability
- Naming conventions
- Architectural conventions

The reuse library at AT&T has proved to be a good base for building powerfully efficient and portable applications. The libraries are written in a subset of C that is compatible with all variants of the C language. Fowler's years of experience have shown him that there is no simple road to building reusable software.

[FRA92] Frakes, William, Ruben Prieto-Diaz, and Edward Comer, "Ada Software Reuse and Domain Analysis," Seminar Briefing, Clarion Plaza Hotel, Orlando, FL, November 16, 1992.

Frakes feels that reusability is a design issue. Designing for reusability must address the scope of potential applications. Frakes proposes that the following criteria should be used in designing new assets and in selecting assets:

- Understandability
- Completeness
- Independence
- Adaptability
- Reliability
- Robustness
- Efficiency
- Portability

Understandability of code rests more with naming conventions and code structure than with comments. Consistent capitalization, underscores, naming conventions make an object's or entity's intended use clear. Application-independent naming with no abbreviations is recommended. Standard headers or prologues are helpful as well as statement comments when additional explanations are necessary. If a life cycle view of assets is taken (e.g., requirements, models, architectures, designs, algorithms, tests), understandability of code assets improves dramatically.

Completeness indicates the reusable software asset provides necessary and sufficient functionality. Values are created, initialized, and default values are

provided. Format conversion and type conversions are supplied. When a state changes, an object's assignment is updated. For composite objects, operations for adding, deleting, iterating, finding, and querying are available. Exceptions are well-formed and test functions exist for every exception that can be raised.

Independence is an indicator of the degree of coupling of modules. Low coupling reduces inter-unit dependencies. High cohesion is permissible with a single function and single abstract data types or objects.

Adaptability must be engineering without sacrificing usability. A reusable component should be sufficiently flexible to promote its reuse, but should not be so extensive as to limit its use. Fewer, simpler interfaces are easier to understand, making a module easier to adapt for other uses. Interfaces should be limited to those specifically required to support the intended degree of reuse. Different mechanism support different degrees of adaptation. Listing lower levels of adaptation to higher levels, they are "as-is," with modification, part families, data/table driven parts, parameterization, generic parts, classes, subsystems, generators constructors, and domain languages.

Reliability is another criteria for designing reusable assets and should answer the following questions:

- What process was applied to the asset (i.e., audits, review, inspections, independent assessments, test or certification)?
- What artifacts of that process would build confidence (i.e., certified algorithms, test procedures, scripts, data, test reports, SPR data)?
- What measurable characteristics of the asset would indicate good reliability (i.e., metrics, reliability models)?
- What else would build confidence in reliability i.e., other usage, particularly in deployed applications, reputation/experience with developing of certifying organizations)?

Robustness is the ability of a component to properly perform in different environments. Reusable components should be designed to query, adapt and conform to its environment or context. The number of environment or contextual assumptions made by a component should be minimized. It is recommended that a component does not explicitly or implicitly interfere with its environment in an unexpected manner.

Efficiency, as well as performance, must be a major design criteria. However, many design and implementation practices that encourage reusability may adversely affect performance and resource utilization. Greater investments in optimization can be made in reusable assets. Understanding of compiler optimizations can ease the tradeoff decisions between reusability and performance.

Portability encapsulates hardware, operating systems, interface software and other implementation dependencies. Design approaches that support reusability are

abstractions, models, layered architectures, and object-based and object-oriented designs.

The qualities that make a component reusable must be engineered into the software. Good software engineering practices alone do not guarantee that the software is reusable. Using the characteristics discussed above will help produce more reusable software.

[HOO91] Hooper, James W. and Rowena O. Chester, *"Software Reuse, Guidelines and Methods,"* Plenum Press, New York, NY, 1991.

The U.S. Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS) initiated and supported a study that led to the compilation of this book by James Hooper. AIRMICS recognized that effective programs for reuse and its supporting technology would be difficult to develop without a book of principles, lessons learned and case studies to guide managers and engineering.

The book provides historical background and introductory information about the problems associated with reuse, its concepts and definitions, research activities, and status of reuse practice. Hooper believes that reuse concepts are moving from research into practice, and very good results are being reported, even with ad hoc processes.

He points out that an initial investment in reuse (i.e., organizational changes, initial library development, training, etc.) is required, and there has been an understandable reluctance to make this investment without reasonable assurance of success. Enough reuse successes are accumulating to allay the concerns; thus he expects an increase in the number of organizations undertaking the practice of software reuse. A number of successes have been based on informal approaches, and may indicate that technical breakthroughs may not be necessary to achieve success in software reuse, although productivity can certainly be further improved.

Hooper notes that there are additional costs when preparing software components for reuse because of the necessary effort to generalize the components, to conduct extra testing, to document the components, and to classify and store them for reuse. An organization must develop a business case to justify the additional cost of developing reusable software. A careful assessment must be made of the likely payoff of such extra costs.

Hooper's book addresses both the managerial aspects and technical aspects of reuse. The final section, "Getting Started," provides guidelines for beginning a reuse program within an organization. Besides the managerial guidelines and the technical guidelines, another important consideration in initiating a reuse program is whether the organization is making use of effective software practices. If would be of little use to attempt a software reuse program without having in place a systematic, consistent process for software development and maintenance. He cites

the SEI's CMM for software as the best-known instrument for such assessments. The software process should be remedied for the organization's inherent benefit and to improve the basis for reuse. He advocates a phased approach for introducing a reuse program to lower its risk of failure. He provides detailed steps and guidelines for this bootstrap process.

[JAC93] Jackelen, George and Larry McCutchan, PRISM Documentation Library, 1.0, Central Archive for Reusable Defense Software (CARDS) ," Software Technology for Adaptable Reliable Systems (STARS), STARS-VC-B007/000/01, December 3, 1993.

This Version Description Document (VDD) describes the long term mission of CARDS as providing operational reusable software libraries designed to support multiple domains. CARDS is also designed to serve as a model or "knowledge blueprint" for the construction of other domain-specific reuse libraries. CARDS consists of the Reuse Library Framework (RLF) and a distributed file system.

[JOH88] Johnson, Ralph E. and Brian Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, June/July 1988, Vol. 1, No. 2, pp. 22-35.

Ralph Johnson believes that even though object-oriented programming is touted as promoting software reuse, it is not necessarily a panacea. For effective reuse, he feels program components must be designed for reusability. He provides a tutorial-like article that describes and organizes a set of design techniques that makes object-oriented software more reusable. Moreover, he feels that as with any design task, designing reusable classes requires judgment, experience and taste.

Polymorphism and inheritance are two features of object-oriented languages that distinguish them from other languages that are based upon abstract data types. Polymorphism increases the likelihood that a given component will be usable in new context. Inheritance promotes the emergence of standard protocols, and allows existing components to be customized.

Frameworks support reuse at a larger level of granularity than classes and allow a collection of objects to serve as a template solution to a class of problems. Johnson defines a framework as a set of classes that embodies an abstract design for solutions to a family of related problems. A framework can be thought of as an object-oriented abstract design for a particular kind of application, and usually consists of a number of classes that may be housed in a library. As a framework becomes more refined, it leads to "black box" components that can be reused without knowing their implementations. Frameworks can be built upon other frameworks.

The product of an object-oriented design is a list of class definitions. Each class has a list of operations that it defines and a list of objects with which its instances

communicate. In addition, each operation has a list of other operations that it will invoke. In order for software developers to rapidly build complicated applications, they must be able to reuse software components and abstract designs that were deigned for reuse.

[KAN89] Kang, Kyo C., "Features Analysis: An Approach to Domain Analysis," Position Papers of the Reuse in Practice Workshop, Software Engineering Institute (SEI), Pittsburgh, PA, July 1989.

Kang suggests that an analysis of the functional features of a system can serve as an approach to domain analysis. The goal of a features analysis is to identify and represent a generalized functional system model from which software requirements can be derived. The generalized model drives the production of software assets and their reuse. His model is also the basis for classification of components and evaluation of their reusability.

His study concluded that there is no adequate mechanism for representing a domain model to support reuse through the requirements phase. Even though no formal approach was followed to arrive at his findings, it appears that future research is needed in this area.

[LEE88] Lee, Kenneth J. et. al., "An OOD Paradigm for Flight Simulators," 2nd edition, Technical Report CMU/SEI-88-TR-30, Software Engineering Institute (SEI), Pittsburgh, PA, September 1988.

Under the Ada Simulator Validation Program (ASVP), a jet engine flight simulator was developed based on theoretical, object-oriented engineering models. Using this paradigm, Lee produced a domain analysis for jet engines. From this, reusable code templates were used to standardize the object interfaces. The templates contained general features of the object while maintaining placeholders for specific object features.

[LEN87] Lenz, Manfred, Hans Albrecht Schmid and Peter F. Wolf, "Software Reuse Through Building Blocks," *IEEE Software*, July 1987.

Lenz cites a quote from T. A. Standish in "Software Reuse," in the Proceedings of the Workshop of Reusability in Programming, ITT, Stratford, Connecticut, 1983, "reusability conditions exist when an application has reached a certain degree of maturity and common abstractions and concept become apparent." Common concepts identify the entities to be reused; when common concepts are not known, a domain analysis is required to identify them.

Even when a concept of a domain has been identified and implemented, it is not necessarily accepted as reusable by its intended users. Several conditions must

be fulfilled before a part qualifies as reusable. It must represent a good modularization with well-selected and usable interface. It must also provide the right degree of functionality. It must provide functional completeness without excessive generality. It must result from good software engineering practices as well as exhibit reusability specific characteristics.

[LUB87] Lubars, Mitchell D., "A Knowledge-Based Design Aid for the Construction of Software Systems," Ph.D. Thesis, University of Illinois at Urbana, Champaign, IL, 1987.

Using a prototypical software design environment, Lubars captured design knowledge and encoded it into representations that abstracted out the common design features across related application domains. His work resulted in design schemas that represent design families with shared similar constraints. These design families can be customized and refined to satisfy a user's requirements. Customization is accomplished by rules for specialization and refinement. Lubars' schema selection strategy facilitates user selection of design fragments. The schema designs provide significant potential for design reuse and can drive asset production.

[MAT84] Matsumoto, Yoshihiro, "Some Experiences in Promoting Reusable Software Presentation in Higher Abstract Levels," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 502-513.

Matsumoto believes that to make software modules reusable, they must have the following characteristics:

1. Generality - the extent to which those who do not know how a software module was developed can understand that module's objects, and the relationships between its objects and algorithms.
2. Definiteness - the degree of clarity to which the module's purpose, capability, constraints, interfaces, and required resources are defined.
3. Transferability - the degree of simplicity in transporting or transferring software between different types of computers.
4. Retrievability - the degree to which a software modules can be selected, stored maintained, and customized by users who have no prior knowledge of its existence.

In order to promote the reuse of existing software modules, Matsumoto proposes the concept of a "presentation." A presentation is a specification of a program accompanied by the ranges in which the project descriptions can be changed when it is reused in another application. Because a presentation describes

an existing program at the highest level of abstraction, (i.e., the requirements level), it provides two benefits:

- Clarity of program behavior - The requirements representation is a direct description of the program's effects.
- Maximization of productivity improvement - The requirements representation abstracts from a larger number of program modules and code fragments than other levels. This promotes reuse at a higher level and may lead to higher levels of productivity.

Matsumoto presents an example of this process as follows. A designer, who plans to develop a new program, P, searches for a presentation which matches P's requirements. If presentation Q matches, program Q', which can be traced back to Q, can be customized to fit P's requirements and will be reused for P.

A requirements description consists of objects, relationships between object, decision-making, input/output transformations, constraints and given facilities. To define a requirements specification for a program, the objects external to the software being developed, and their relationships to the objects internally, are defined. The types, attributes, and the relationships associated with each external object are specified. Then, the states of the external object are defined. When an object in one state moves into a new state, an event occurs. Subsequently, a decision-making process may be activated in order to select the next action.

After the requirements description, the second level of abstraction or the data/function or design level, is completed. Data structures, functions, data flows, and control flows are defined in this phase.

The third level of abstraction is called the program level, the transition from design to programming-in-the-large domain. The external structure of program modules are designed. Program configuration, file structures, and package interfaces are created using data flows, data structure, function, and control flows. Resources are optimized to satisfy given constraints and obtain the best performance. Decompositions and integration are repeated until acceptable functional configurations and file structures are obtained.

Continuing at the program level, the large program structure is planned as a result of the above processes. Real-time tasks, packages and subprograms are determined. Package specifications and internal structures for package and data structures are designed. Traceability is verified by comparing descriptions from the early phases of this process to the final implementations. In order to increase reusability of specific packages, they are rewritten in a generalized format.

The entire presentation and all its artifacts are stored and later used when specifying a new problem and its solution. This method has been highly successful in software manufacturing applications for real-time process control systems. Matsumoto's company, Toshiba, averages four million lines of equivalent

assembler code per month, with 3000 employees in their software factory. The reuse rate for their software products is about 50%.

[MCC85] McCain, Ron, "A Software Development Methodology for Reusable Components," *Proceedings of the Software Technology for Adaptable Reliable Systems (STARS) Workshop*, pp. 361-384, Naval Research Laboratory, Washington, D.C., April 1985.

Many papers on reusability have focused on the need for reusing software and a component library as a means for accomplishing reuse. Instead, McCain focuses on how the component should be constructed for reuse, that is, designing for reuse during software development. McCain feels that unless the software industry adequately establishes software development approaches that emphasize the construction of reusable software components, attempts to reuse components for a software library will be futile.

McCain states that for components to be reusable, they must be "useful" and "usable." A reusable component is useful if it is applicable to multiple users. Potential reuse can be maximized by developing components that have a substantial domain of applicability. A reusable component must also be usable. Even if a component has substantial applicability within and across domains, unless it is usable, it is not a good candidate for reuse. Factors that affect usability are specification precision, user knowledge proximity, interface abstractness and functional cohesion.

McCain presents a candidate methodology for the development of reusable components. His method can be used for customized development as well as general development and at all stages of software decomposition. McCain's method uses these steps to develop reusable software components:

1. Define the interfaces - Interfaces must be completely and accurately specified.
2. Limit dependencies - The component must have minimum dependency on other components.
3. Perform domain analysis - For the current specification, identify users and their needs, identify reusability constraints of the specification, identify commonality across domains, and abstract for maximum domain reuse and extended domains.
4. Reuse existing software if available - Determine if it is cost-effective to reuse software to satisfy the requirements. If so, then reuse. Otherwise, proceed with new development with the goal of producing reusable software. If only a portion of the software can be reused, define a new specification that reflects the current component, specify the requirements not accommodated by reuse, and then repeat the process.

5. Define the current specification of the reusable object and its operations.
6. Define the current specification of reusable abstractions. Layered abstractions encourage reuse.
7. Define an abstract interface specification and its reusable abstractions.

A formal way to validate the implementation of reusability during development involves evaluating applicable existing software, a domain analysis summary, an abstract interface specification, and an abstract constraint analysis summary. Reusability assessment is recommended as part of the review process and should include component programmers, domain analysts, software component engineers, and component users.

In order to dramatically reduce software cost, software developers need to learn how to reuse existing components. To accomplish this, they must first learn to develop components to be reused. By examining characteristics of reusable software components and establishing a method that allows components to be constructed with these characteristics, an initial step has been taken to influence the production of reusable software components. McCain's method needs to be validated and enhanced by applying it to pilot projects. Work production enforcement mechanisms and support tools must be put in place to make his method a part of normal software development.

[MEY87] Meyer, Bertrand, "Reusability: The Case for Object-Oriented Design," ***IEEE Software,*** **March 1987. pp. 50-63.**

Bertrand Meyer maintains that the fundamental goal of software engineering is reusability, and its companion requirement, extendibility (i.e., the ease with which software can be modified to reflect changes in specifications.) Progress in one of these areas usually advances the aims of the other. He feels that object-oriented design is the most promising technique known for attaining the goals of reusability.

Meyer acknowledges that some reasons why reuse isn't more common can be categorized as non-technical (e.g., few economic incentives, not-invented-here complex, lack of libraries or reusable modules and good database search tools so programmers can find appropriate modules easily). However, he believes that non-technical issues are only the tip of the iceberg; reuse is limited because designing reusable software is difficult.

The purpose of his article is to dispel the naive hope that software problems would just go away if we were more organized in filing program units. One estimate is that less than 15% of new code serves an original purpose. He contends that programmers do tend to do the same kinds of thinking time and time again, but they are not exactly the same things. So many details may change as to render

moot any simple-minded attempt at capturing commonality. He quotes Gerard de Nerval, "neither ever quite the same, nor ever quite another."

Even though the patterns for particular algorithms may be standard, the amount of variable information is considerable. It is difficult to implement a general purpose module; it is almost as hard to specify such a module so that dependent modules can rely on it without knowing the implementation. Beyond the basic problem of factoring out the parts that are common to all implementations of a function, an even tougher challenge is to capture the commonality without some conceptual subset. He feels that to write carefully organized libraries of reusable software elements, we must be able to use commonalties at all levels of abstraction.

Several approaches have been used to solve the reusability problem. For example, the classical technique is to build libraries of routines (i.e., procedures, function, subroutine, or subprograms) that implement a well-defined operation. This approach has been quite successful in scientific computation, and excellent libraries exist for numerical applications. The library of routines seems to work well in areas where a set of individual distinct problems can be identified. These problems have a small set of parameters and complex data structures are not involved.

For more complex problems, higher level languages (e.g., Ada) provide higher structuring than a routine. This approach is rooted in the theory of data abstraction. The techniques of overloading or genericity allows a module to be defined with generic parameters that represent types. Instances of the module are then produced by supplying different types as actual parameters. This is a definite boost to reusability because just one generic module is defined, instead of a group of modules that differ only in the types of objects they manipulate. However, these techniques alone do not provide enough flexibility and forces programmers to decide too much too soon.

Meyer views object-oriented design as a software decomposition technique. Object-oriented design bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions that system performs. He feels that it is wiser to rely on categories of objects as a basis for decomposition, but only if these categories are viewed at a sufficiently high level of abstraction. Object-oriented design differs from a top-down functional approach that solves a fixed problem once and for all. Object-oriented techniques accommodates a long-term view for long-lived systems.

Object-oriented design also relies on abstract data types which describe a class of objects through the external properties of these object instead of their computer representation. An abstract data type is a class of objects characterized by the operations available on them and the abstract properties of these operations. Abstract data types are useful at the design and implementation stage. Object-oriented design is the construction of software systems as structured collections of abstract data-type implementations. A single program structure is both a module

and a type, dubbed as a "class" by the creators of the pioneer object-oriented language, Simula 67. Instances of classes can inherit generic functions from its parent, yet can also be specialized to met its particular requirements. These object-oriented techniques, classes, abstract data types, inheritance, and instances all enable software reusability and extensibility.

[NEI80] Neighbors, James, "Software Construction Using Components," Ph.D. Thesis, University of California at Irvine, CA, 1980.

Neighbors asserts that the optimal software reuse is through reuse of analyses, designs, and code, rather than simply from the reuse of code. He introduced the concept of domain analysis to describe the activity of identifying objects and operations of a class of similar systems in a problem domain.

[NEI83] Neighbors, James, "The DRACO Approach to Constructing Software from Reusable Components," *Proceedings of the Workshop on Reusability in Programming,* ITT Programming, Stratford, CT, September 1983, pp. 167-178.

Neighbors' goal of his DRACO approach to constructing software was to increase productivity of software specialists in developing similar systems within a problem domain. The first DRACO prototype was completed in 1979 and the last major revision of the mechanism was completed in 1983.

The DRACO approach to the construction of software from reusable software components focuses only on the constructive aspects of software production (i.e., analysis, design, implementation). It does not address with the organizational interactions of development team members or methods for the complete specification of software systems. Neighbors believes that the reuse of analysis information is the most powerful kind of reuse. The reuse of design information is the second most powerful kind of reuse. Consequently, DRACO captures the expertise of an organization and delivers it in problem-specific terms.

[PAR76] Parnas, David, "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering,* Vol. SE-2, No. 1, March 1976, pp. 1-9.

Parnas defined program families as sets of programs whose common properties are so extensive that it is advantageous to study those before analyzing individual members. His early work paved the way for the development of domain analysis, reuse and architectures as an asset type for reuse.

[PAR79] Parnas, David, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering,* Vol. SE-5, No. 2, March 1979, pp. 128-138.

Parnas identified minimal subsets and minimal extensions during software design which leads to software that can be tailored to the needs of a broad variety of users.

[PAR85] Parnas, David, Paul C. Clements and David Wise, "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 3, March 1985, pp. 259-266.

Parnas and his team created a guide to modular structuring using an example from the domain of the Operational Flight Program (OFP) for the a-7E aircraft. He found that the software engineering principle of information hiding is practical for complex systems. Documenting this guide was useful to designers and programmers in resolving design and communications problems. The guide was also helpful for training new staff on the project facilitating their understanding of the structure of the program.

[PET93] Petracca, Aleisa, Les Hayhurst and George Jachelen, "Portable, Reusable, Integrated Software Modules (PRISM) Documentation Library Model, Document Release 1.0, Central Archive for Reusable Defense Software (CARDS), Software Technology for Adaptable Reliable Systems (STARS), STARS-VC-B015/000/00, December 3, 1993.

The CARDS (Central Archive for Reusable Defense Software) program believes that a formal, systematic integration of reuse into the conventional software development process yields the greatest reward. Petracca shows how domain analysis, within the field of Domain Engineering, is a technique that can help integrate reuse into the conventional software development process. This Informal Technical Report from CARDS describes modeling concepts and examines the principles of specialization and aggregation using hierarchies.

Domain analysis describes the requirements for a family of systems (i.e., establishes the requirements of a domain). In the field of Domain Engineering, domain analysis is similar to requirements analysis in the field of Software Engineering. Likewise, a generic architecture specification in domain engineering is similar to the activity of system specification in software engineering. The system implementation in software engineering is analogous to the generic architecture implementation in domain engineering and forms the foundation for the parts library. CARDS envisions the parts library as serving to back fill the development of a specification and its implementation in future systems.

CARDS is envisioned to house components of many domains, but has initially been targeted toward command centers. CARDS relies on the program PRISM (Portable, Reusable, Integrated Software Modules) as its primary source for information about the domain of command centers. The information consists of its models, components, evolution, and documentation. Within CARDS, the Reuse Library Framework (RLF) is the mechanism used to implement modeling. The RLF has three parts; a knowledge-representation schema called AdaKNET, a rule-based inferencing engine; and a graphical browser.

The five year plan for PRISM is to serve as a management tool to assist staff in the activities of identifying and documenting critical project objectives and associated dates and milestones. PRISM assists in detailed demonstration planning, resource identification and allocation, and tracking of future technology trends. This purview should supply CARDS with many components from the domain of command centers. The document also describes a procedure for qualifying software components for incorporation into the generic command center and the generation of product assessment reports.

[SIM87] Simos, Mark A., "The Domain-Oriented Software Life Cycle: Towards an Extended Process Model for Reusability," ***Proceedings of the Workshop of Software Reuse,*** **Rocky Mountain Institute of Software Engineering, Boulder, CO, October 1987.**

Simos proposed that reusability needs to be integrated into the conventional top-down "waterfall" life cycle model of software development. This life cycle development model, extended for reuse, should have the following components:

1. A perspective centered upon domains or families of related program or systems that support particular application areas
2. Concentration on application specificity, or narrow-band reuse within specific application domains
3. Recognition of a set of techniques for reusable software, either through ad hoc reuse, libraries, code generation techniques, and/or knowledge-based techniques

His project model can also be used to target resources for productivity increases as well as identify traceability across projects.

[TRA87] Tracz, Will, "Reusability Comes of Age, ***IEEE Software,*** **July 1987, pp. 6-8.**

Tracz answers the question "What will it take to create a successfully used program?" He uses the analogy that compares used cars to used programs; he believes that this analogy holds for pungent and practical reasons. People are leery

about buying a used car for many of the same reasons programmers are reluctant to reuse someone else's work. He views reuse as a study in sales of used-programs.

Users are interested in the following characteristics:

- Quality parts - Customers should have confidence that what they buy will perform without error.
- Standard interfaces - Customers should be able to use what they buy in a manner that complies with standard operation conventions. Software should be easily integrated into new or existing systems.
- Documentation - Customers should understand what the software does, how they use it, and how they can modify it if necessary.
- Selection - Customer should have a choice of options available for what they buy.

Tracz sees reuse as a business with customers to satisfy. Even though it may be an oversimplification of the problem, his fundamental principles are worthwhile.

[TRA93] Trail, Glen and George Jachelen, "Portable, Reusable, Integrated Software Modules (PRISM) Documentation of Library User's Guide," Release 1.0, Central Archive for Reusable Defense Software (CARDS), Information Technical Report from the Software Technology for Adaptable Reliable Systems (STARS) Program, STARS-VC-B006/001/101, December 3, 1993.

The Generic Command Center (GCC) project, the forerunner of the Portable, Reusable, Integrated Software Modules (PRISM) project, integrates components for use in command centers. PRISM succeeded GCC in January 1992, and has the goal of providing details for a generic architecture for command centers. A program description language specifies the architecture and initial automated support is provided through PDL (Program Description Language) Model Release 1.0. This user guide describes how to access the automated tool for this architectural model.

PRISM proposes to supply users with 80% of the required resources to produce a new command center as well as the information on acquiring or producing the remaining 20%. The PDL model can be run remotely through a modem.

[WAR88] Ward, Paul T. and Lloyd G. Williams, "Using the Structured Techniques to Support Software Reuse," *Proceedings of the Structured Development Forum,* San Francisco, CA, August 1989. pp. 211-222.

Ward feels that object-oriented development and domain analysis are two techniques which offer support for reusable components. Object-oriented techniques provide structure to components providing the potential for reuse.

Domain analysis assists in identifying components that should be designed and produced with reuse as a goal.

[WEI88] Weiss, David, "Reuse and Prototyping: A Methodology," Technical Report SPC TR-88-022, Software Productivity Consortium (SPC), Reston, VA, March 1988.

Weiss proposed a method for software development and maintenance that encompasses prototyping and reuse. The underlying concept of his method is based upon information hiding, program families, hierarchical structuring, and characterization of modules as black boxes. Tools can be built to search through a collection of program families, adapt family components to create new family members, and compose new family members from existing components. The tools can be used to describe, assess, and store families, including information needed to characterize them for future use.

Appendix D - Annotated Bibliography of Asset Selection

The information gleaned from this literature survey of asset selection was used to determine the operational context of the Reuse Context for Asset Quality Certification and to assess the impact of this previous research on the development of the Certification Framework.

The annotations in this appendix summarize the essence of each of the referenced publications. Summaries vary in length; those that are longer provide additional details because the reference appeared to be a flagship among others. The shorter annotations were still included to serve as a pointer to the complete reference if more details are of interest.

This annotated bibliography this area is not exhaustive, but gives a flavor of the previous research that has been accomplished. Some of these references were used in other appendices.

[ARA95] Arango, Guillermo, "Software reusability and the Internet, " ***Proceedings of the Symposium on Software Reusability , SSR'95,*** **edited by Mansur Samadzadeh and Mansour Zand, Seattle, WA, April 28-30, 1995.**

In an briefing given by Guillermo Arango at the Software Reusability Symposium, he maintains that the world-wide communication infrastructure of the Internet could be the best thing that has happened to the software reusability community in the past twenty years. He believes the Internet will provide a megalibrary where people and resources can meet and exchange products and services.

The scale of resources becoming available to software developers, the ability to share expertise in a global market, and new business practices are defining a different environment for software reuse. These changes do not affect the problems inherent of the past, but it does change the environment in which the problems reside.

He admits that some may argue that in cases involving industrial software, we cannot risk reusing "flaky stuff" from the Internet. This is an issue, based on our experience of the Internet as it works today, but it is a symptom of the lack of maturity of the legal framework and of the business practices in software electronic commerce. He does not believe that this is a truth about the potential of the Internet as a distributed megalibrary.

Standards and processes will be critical for his vision to become practical. Arango believes that the software industry is better positioned to examine all the technical and professional issues involved. He believes that the reusability community has a historic opportunity to take the initiative in making a substantial contribution to the field of software engineering.

[ARM95] U.S. Army Space and Strategic Defense Command Software Engineering Division, "Component Evaluation Procedure (Phase II) Technical Report, January 31, 1995.

This technical report states that the DoD software costs are expected to reach $42 Billion in 1995. The DoD Software Reuse Initiative believes that effective reuse can contribute to the reduction in growth of software costs while providing improved system performance and reliability. This document describes the Components Evaluation Procedure (CEP), being developed by the Software Engineering Division (SED) at the U.S. Army Space and Strategic Defense Command, Huntsville, AL. The CEP is seen as a way to provide cost-effective, quantitative measurement of the reusability of existing software components.

The CEP can be applied to all types of software components (e.g., requirements, designs, documentation, and test data). However, current efforts have concentrated on applying the CEP to Ada source code components.

The CEP consists of three principal elements, that is, criteria application, risk analysis and report generation, as shown in Figure D-1. A model of the CEP is shown as a process flow diagram with subactivities within each principal element.

Four phases of CEP development were planned and two have been completed to date:

1. Phase I - resulted in the CEP model and a list of twenty-one proposed reusability criteria, subdivided into four categories. Late in Phase I, these were integrated with a list of thirteen reusability criteria from an ongoing project at the program office.
2. Phase II - applied and validated the set of reusability criteria using Ada components from the Battle Management/Command, Control, and Communication (BM/C^3) domain. As of January 1995, the CEP development was in this Phase II.
3. Phase III - plans to develop risk analysis and report generation elements of CEP.
4. Phase IV - plans to validate the CEP.

The findings of Phase II were identified as follows:

- Interdependency of Ada compilation units (i.e., the number of "with"s) is a major inhibitor of reuse.
- Domain applicability greatly affects component reusability. This supports the view that reuse should be domain-specific.
- Measures of component size showed moderate to weak statistical correlation with reusability.
- It appears to be worthwhile to screen potential reusable components for good software engineering characteristics using automated tools (e.g., quality, maintainability). The use of automated tools makes screening a relatively inexpensive process.
- Based on the results of this activity, no easily measured structural characteristic of an Ada source code component can be used as an effective indicator of that component's reusability in a specific application.

These findings were generated from ratings and measures collected from two parallel activities of application and validation, those of reuse experts and those of software engineers.

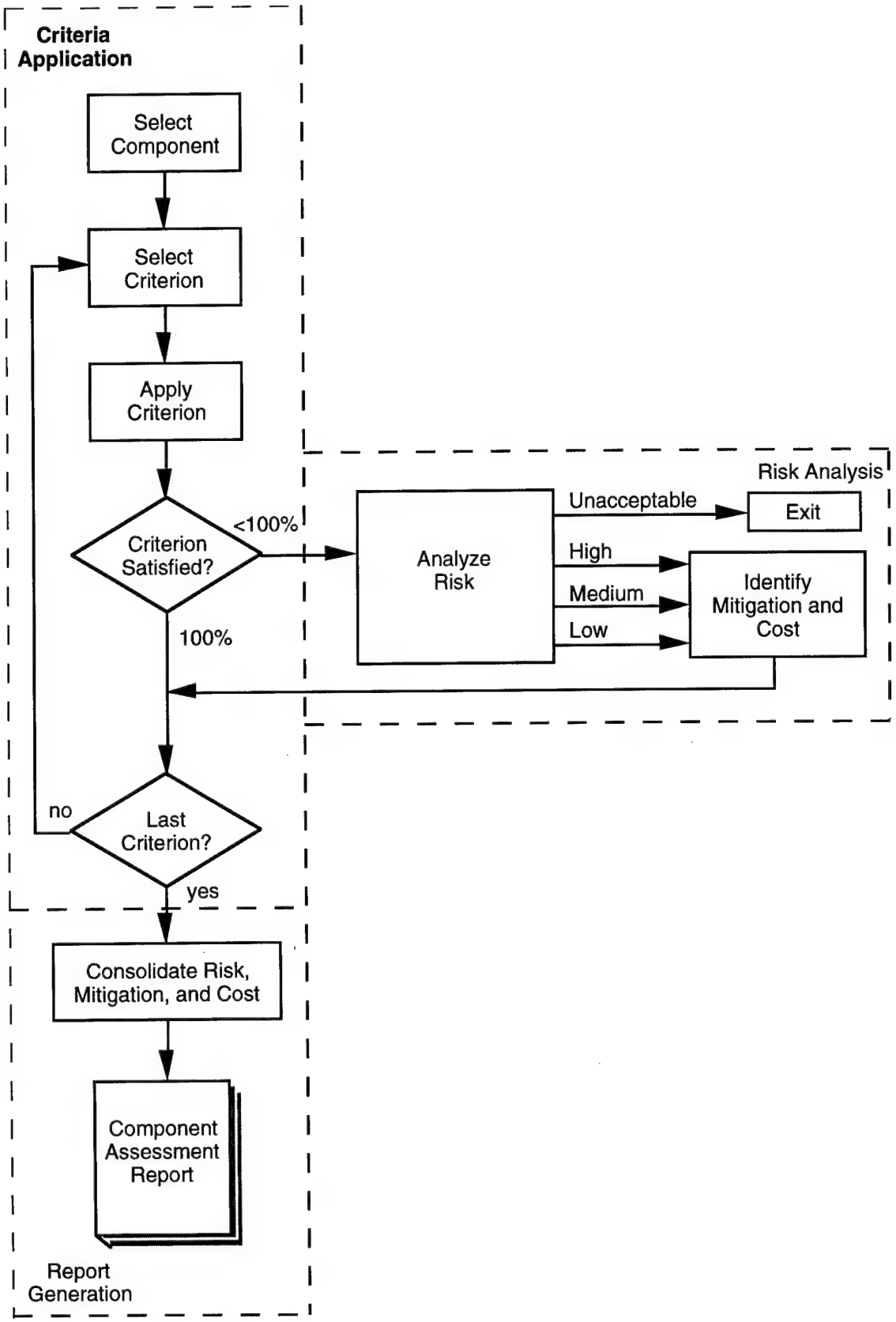


Figure D-1. CEP (Components Evaluation Procedure) Model [ARM95]

The technical report also cites work related to the CEP. For example, the DISA/JIEO/CIM's Software Reuse Metric Plan proposed the use of AdaMat to predict component reusability. If the organization was able to validate this proposed concept, it would aid reusers in selecting assets. To date, this concept has proven inconclusive.

Also related to the CEP is the DSRS. DSRS addresses administrative suitability for reuse, but does not attempt to predict the operational quality or reusability of components. Instead, the DSRS's certification procedure for software components evaluates completeness of documentation and conformity of a component's behavior to its functional description.

Another work related to the CEP is the program called PRISM (Portable, Reusable, Integrated Software Modules) managed by the U.S. Air Force Electronic Systems Center (ESC). PRISM is chartered to develop a reusable generic command center software architecture to reduce time and cost of acquiring command centers. PRISM has adopted the library concept of the Comprehensive Approach to Reusable Defense Software (CARDS) as the basis for its domain-specific software repository (i.e., Command and Control (C^2)).

PRISM identified reusability criteria (e.g., adaptability, domain applicability, documentation, simplicity and readability, complexity) and outlined procedures to collect, both automatically and manually, quantitative measures for each. The following findings were reported:

- The levels of reusability were associated with components with fewer than four "with" statements (i.e., components with eight with statements scored poorly in reusability).
- Applicability of a component to a domain is highly correlated to reusability.
- Simplicity and readability impact the ease of understanding which is a prerequisite to evaluating reusability.
- Complexity, as related to simplicity and readability, is a useful indicator of reusability.

These findings were collected from a data set of the EVPA (Experimental Version Performance Assessment), a large, distributed simulation designed to support the testing and integration of strategy defense software. EVPA has a set of legacy Ada components that are representative of much of the existing Government software (i.e., poorly documented and the original authors are no longer available.)

[BAN93] Banker, Rajiv D., Robert J. Kauffman, and Dani Zweig, "Repository Evaluation of Software Reuse," ***IEEE Transactions on Software Engineering,*** **Vol. 19, No. 4, April 1993.**

Rajiv Banker found that the probability that a programmer will reuse an existing software object rather than write a new one depends upon the availability of potentially reusable software and upon the programmer's ability to find it. However, he also found that reuse did not necessarily grow as the pool of reusable candidates grew. He observed that reuse is also driven by a pool of familiar code rather than the entire pool of reuse candidates. Familiar objects within the programmer's domain are more likely to be reused, regardless of the size of the pool of candidates.

Furthermore, he reports that programmers have individual differences with regard to their practice of reuse. A small number of outstanding programmers appear to account for a disproportionate amount of reuse. Some of the same skills that make some programmers extraordinarily productive also make them extraordinarily good at reuse. Consequently, he concludes that teaching these skills could promote reuse.

[BER95] Bergstrom, Deane, "Certification of Reusable Software Components," Briefing chart in response to Project Overview, December 12, 1995, Rome Laboratory, NY.

In response to a project overview given by Software Productivity Solutions, Inc. (SPS), Deane Bergstrom prepared and presented two briefing slides to describe the context for the project and considerations for the framework as shown in Figure D-2. This concept flows down from the SOW requirements for the certification framework.

The context is defined by the expected use and user profile, the range of capabilities to be provided, the product inventory, and the interfaces to the users and sustainers of the process and environment.

The certification framework must define the context to be included or account for those capabilities that are covered in the library. The coupling between the reuse library and the certification framework may be loose or tight, joined by a template contained in the user interface for data interchange between the library and the certification framework.

More importantly, the software developer must be able to perform multi-pass browsing of the assets to support several phases of survey, examination and selection. The component selected may be one, few, many or an entire system. The developer may want existence proof of required artifacts and may need to quantify relationships of current components in library systems. The framework must also support resource constraints of schedule, cost, functional capability, performance, skill levels and size.

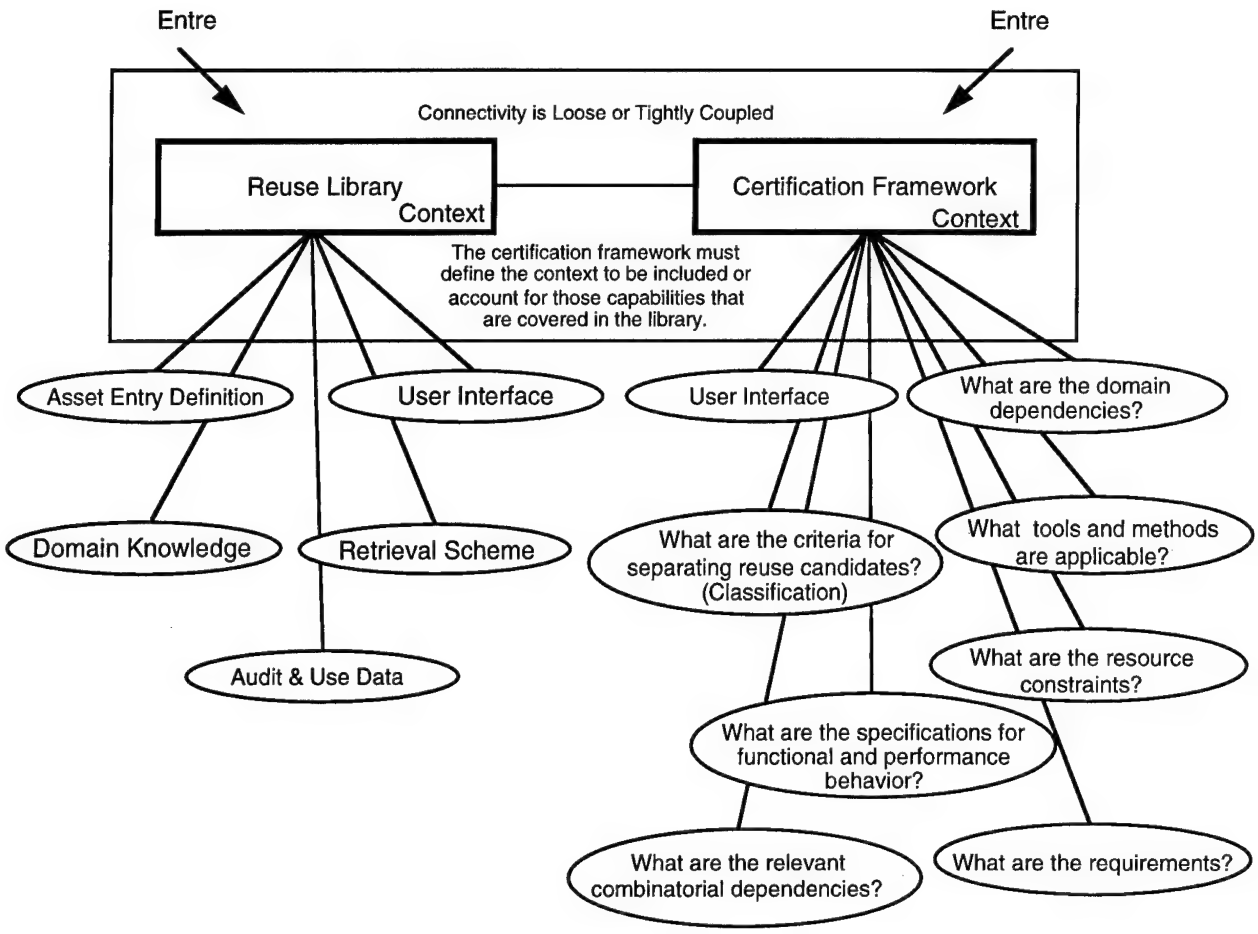


Figure D-2. Certification of reusable software components [BER95]

The user's needs may be know, not known, or yet to be determined based on use and experience. The framework must accommodate a steep learning curve for the user community. "Make/buy" decisions need to be simplified and users may need data for constructing a justification for their selection. Typical questions that the reuse library may ask are the following:

- What information and/or data can I get from the context of the reuse library schemes in use that will assist in defining the certification process?
- What (if any) is the common subset of reuse library characteristics and what are the skill levels of the range of users?

These types of questions and the conceptual view of the reuse library and certification framework helped drive the project development to its completion.

[DIS95] Defense Information Systems Agency Center for Software, DoD Software Reuse Initiative, "Software Reuse Business Model (SRBM)," Technical Report, January 31, 1995.

In the public library archetype of the Software Reuse Business Model (SRBM), assets are considered "free." However, it is not a "no cost" proposition to reusers. One of the activities in the role of the Program Manager as identified in the SRBM is to identify the process of reusable asset selection. In addition, the reusers must still accept the cost and risk of searching the library for suitable assets, modifying the assets to their needs, if necessary, and verifying their applicability. Both these tasks may not be trivial.

[FIS87] Fischer, Gerhard, "Cognitive View of Reuse and Redesign," *IEEE Software*, July 1987.

Fischer maintains that the reuser needs better support in creating new systems (through reuse) and modifying existing ones (through redesign). He feels that software environments must support design methods whose main activity is not only generating new programs but also maintaining, integrating, modifying, and explaining existing ones. Incremental and evolutionary reuse and redesign must be efficiently supported for ill-structured problem domains.

New architectures and intelligent support tools are needed to reduce the cognitive demands that innovative technology has brought. These tools must support incremental learning and learning on demand, two prerequisites for reuse and redesign. These intelligent support tools must also be able to volunteer help in appropriate situations rather than respond solely to explicit requests.

Fischer believes that several cognitive problems prevent users from successfully exploiting their function-rich systems. Users do not know what building blocks and support systems exist, when to use these tools, what the tools do, and how to combine, adapt, and modify tools to their specific needs.

In order to translate the problem of the situation into a previously existing model, reuse is required to construct the right sequence of operations to yield the solution. To do this, search strategies must be used. Building blocks are useless unless the designer knows that they are available and how the right one can be found. Psychological research has shown great differences exist between the efficient and successful strategies used by experts and the inefficient and ineffective search strategies of novices.

Knowing about the existence of components is not trivial, especially as the number of components grows. And if a reuser does find a potentially useful component, he must determine how it must be used and combined with the other components. He must understand its functionality and its properties.

Fischer's observations of designers is that they do not engage in reuse and redesign because these methods are not adequately supported. The effort to change a system or to explore design alternatives is too expensive in most production environments. If the cost of making changes is cheap enough, users will start to experiment to gain experience and insights leading to better designs. Fischer believes that much is wasted because users do not understand how to the use the software industry's full potential.

[LAT89] Latour, Larry, "Issues Involved in the Content and Organization of Software Component Information Bases, Interim Report," Technical Report for the U.S. Army CECOM, prepared by the University of Maine, Orono, ME, May 1989.

Larry Latour maintains that the primary inhibitor to the reuse of software components is understanding. He investigated new and innovative techniques for organizing a database of reusable components. Hypertext was used as a tool to describe taxonomies of Ada Booch packages to facilitate reuse and asset selection. He defined a component as an information "web" of attributes containing a specification, an implementation and usage information.

[MCN86] McNicholl, Daniel G., et. al. "Common Ada Missile Packages (CAMP) Vol. 1, Overview and Commonality Study Results," Technical Report AFATL-TR-85-93, McDonnell Douglas Astronautics Company, St. Louis, MO, May 1986.

Daniel McNicholl developed an associated parts catalogue schema and parts composition system to support software parts usage and asset selection. He studied commonalities within the domain of missile flight software systems to drive the development of reuse parts for that domain. Two phases of the project were defined; the concepts were identified in CAMP-1 and then implemented in CAMP-2. When CAMP-2 was completed, 454 production-quality reusable Ada parts were coded, tested, and documented in accordance with 2167A. A parts composition tools was demonstrated which generated Ada code for user-specified subsystems.

[POO92] Poore, J.H., Theresa Pepin, Murali Sitaraman, Frances L. Van Scoy, "Criteria & Implementation Procedures for Evaluation of Reusable Software Engineering Assets," Software Technology For Adaptable, Reliable Systems (STARS) Program, Task/Subtask IT00.19, CDRL Sequence 04014-002B, July 16, 1992.

The purpose of this report was to formulate criteria and procedures for the evaluation of reusable assets in the context of the ASSET reuse library. Four levels of quality are defined for assets; Level 1 (Documented), Level 2 (Audited), Level 3,

(Validated), and Level 4 (Certified). Level 4 (Certified) means that ASSET has conducted an independent, repeatable, formal evaluation according to a predetermined and published protocol and certifies the asset according to that protocol. This certification will, in most cases, require expertise beyond ASSETS' own staff. Evaluation criteria are established with a focus on the quality or an asset, or how well an asset does what its supplier claims it is supposed to do.

Certification protocols for a code type of assets could be dynamic (based on testing), or static (based on proof). Dynamic certification would entail the following activities:

a) Construct a model of intended use of the software.

b) Randomly generate a sufficient number of test cases based on the usage model to certify at the desired level.

c) Execute the code with the test cases and compare actual behavior and performance with the specification.

d) If a deviation from the specification is encountered, confirm this with the supplier and delete the asset from the library.

e) If a sufficient number of tests are passed without failure, the code is certified at some stated confidence.

Static certification would entail verification by abstracting the behavior of the code or constructing a written proof that the function computed by the code is equivalent to the intended function of the code.

In the case of certifying documents, ASSET staff should attempt to independently duplicate the experience conveyed by the document.

[POU95] Poulin, Jeffrey S. and Keith J. Werkman, "Melding Structure Abstracts and the World Wide Web for Retrieval of Reusable Components," ***Proceedings of the Symposium on Software Reusability (SSR'95),*** **edited by Mansur Samadzadeh and Mansour Zand, Seattle, WA, April 28-30, 1995.**

Jeffrey Poulin believes that reusable software libraries have largely failed to return the reuse benefits promised by their developers because they suffer from poor interfaces, too many formal standards, high levels of training required for their use, and most of all, a high cost to build and maintain. Poulin describes an implementation of a reusable software library at Loral Federal Systems using the World Wide Web (WWW) browser Mosaic and shows how it meets most user needs, avoids pitfalls and costs only a fraction of the costs of more traditional libraries.

Poulin named Loral's software reuse library interface the Federal Reuse Repository (FRR). The FRR provides three ways to locate a needed component:

1. Hierarchical view
2. Subject listing
3. Keyword search

The first way, the hierarchical view, narrows the search based on implementation language or sublibraries. If language is not important to the user, or he does not know the component's sublibrary, he can search, by subject, for components that perform a particular function. And lastly, his keyword search supports Boolean queries and partial matches while ranking the results to help the user determine which components must closely meet his needs.

Poulin believes that elaborate classification schemas with facets and attribute values do not give the user an intuitive feel for the applicability of a particular module to a specific situation. He believes that mapping the user's idea of what he needs to an existing component must happen quickly, efficiently, and painlessly. The reuser needs to make the most of those first precious seconds and make the best possible reuse decision. Consequently, he developed a techniques called the Structured Abstract (SA).

The SA quickly provides the user the most needed reuse information. Using this natural-language abstract presents the information to the user in a familiar way, mimicking the manner the user would receive the information from a colleague over the phone or in a conversation. Poulin feels that his method has been successful at Loral because it is natural to the user.

The SA contains the following items:

- Computer language and component type
- Domain
- Function
- Data
- Operating System
- Element
- Contact

A template for this information as it appears in the FRR looks like this:

A (*Computer Language*) (*Component Type*) for (*Domain*) that provide (*Functions*) on (*Data*) data. Runs on (*Operating System*) includes (*Element, ..., Element*) Contact (*Contact*).

Since 1994, numerous repositories for all types of information have emerged on the WWW. One of these, GAMS, at the National Institute of Standards and Technology (NIST), provides an on-line cross-index of available mathematical modeling and statistical analysis software. GAMS differs from FRR in that its keyword fields are structured and only provides independent attribute-value pairs

rather than a coordinated, textual description of a component as in the SA. The SA and the FRR were developed at 1% of the cost to develop a standard software reuse library. The FRR has also gained favor due to its intuitive interface and simple, yet powerful, information retrieval tools.

[PRI87b] Prieto-Diaz, Ruben, "Faceted Classification and Reuse Across Domains," *Proceedings of the Workshop on Software Reuse,* Rocky Mountain Institute of Software Engineering, Boulder, CO, October, 1987.

Ruben Prieto-Diaz proposed an approach to facilitate reuse and asset selection across domains using domain analysis and faceted classification. Domain analysis derives faceted classification schemes of domain specific collections, then derives a global faceted scheme that relates the different domain-specific vocabularies. A global scheme allows users to identify and select components from different application domains. Prieto-Diaz believes that using these technologies for asset selection increases the potential reusability of components.

[TRA87] Tracz, Will, "Reusability Comes of Age, *IEEE Software,* July 1987, pp. 6-8.

Will Tracz uses the analogy that compares used cars to used programs in this article and others in *IEEE Computer* of April 1983, June 1986, and May 1987. He maintains that people are leery about buying a used car for many of the same reasons programmers are reluctant to reuse someone else's work.

Tracz explores several aspects of asset selection using the "used car" analogy:

1. New or used? - Before deciding on whether to invest in a new or used car, a prospective buyer first identifies his needs. He must determine the features, performance, price range, urgency, etc. that are best for him.
2. Standard features - Does the car meet the user's requirements? The features, performance, expected maintenance, and the total price all have their particular tradeoffs.
3. Mileage - A low mileage car may lead to suspicion, whereas in software, high mileage (i.e., increased users) is desirable. In software, the number of bugs can decrease with use.
4. Maintenance record - The types of repairs for cars and software may influence the decision to buy or use it. The prospective user would do well to examine the history of repairs (i.e., Were repairs performed early in its life? What type of repairs were made? What was the severity of the problems?). Such an analysis may avoid a situation where more problems were introduced with each repair.

5. Reputation - The character of the manufacturing organization is usually inherent in its products for both cars and software.
6. Appearance - How does it appear on the exterior and under the hood?
7. Standards - Are there standard seat belts and emission controls on the car? Likewise, does the software have defined requirements, interface designs, adequate testing and clear documentation that follow industry standards?
8. Warranty - Do both products have credibility and viability?
9. Modification and customization - Has the car or software ever been modified and customized? What were the results of that activity?
10. Options and Associated Risks - What are the options and risks for each product?
11. Accessibility - Is the producer of the product available for product support?
12. Price - Options (desired and undesired) may be packaged together and affect the total price. Will the user need training to operate and maintain the product as is, and with its available options? How does this affect the total price?
13. Test drive - This is the "acid" test to determine the suitability of the product for the user. The test drive should simulate multiple working conditions that adequately demonstrate suitability (i.e., usage scenarios).
14. Intangible inhibitors - What is the reputation of the seller? Is there a risk of "getting stuck with a lemon?"

By exploring these topics, Tracz discusses how to create a successful used-program business.

Appendix E - Annotated Bibliography of Reuse Frameworks

The information gleaned from this literature survey of reuse frameworks was used to determine the operational context of the Reuse Context for Asset Quality Certification and to assess the impact of this previous research on the development of the Certification Framework.

The annotations in this appendix summarize the essence of each of the referenced publications. Summaries vary in length; those that are longer provide additional details because the reference appeared to be a flagship among others. The shorter annotations were still included to serve as a pointer to the complete reference if more details are of interest.

This annotated bibliography this area is not exhaustive, but gives a flavor of the previous research that has been accomplished. Some of these references were used in other appendices.

[BIG89] **Biggerstaff, Ted. J and Alan J. Perlis, *"Software Reusability, Volume I, Concepts and Models* and *Volume II, Applications and Experience,"* ACM Press New York, NY, 1989.**

Biggerstaff and Perlis are the editors of this two volume compendium of work by distinguished researchers in software reusability. In the first article, Biggerstaff and Charles Richter define a reusability framework that describes reusability from a technology point of view. The technologies that are applied to the reusability problem can be divided into two major groups depending upon the nature of the components being reused. These groups are composition technologies and generation technologies. Table E-1 shows their framework for classifying the available technologies.

Table E-1. A framework for reusability technologies [BIG89]

Features	**Approaches to Reusability**				
Component Used	Building Blocks		Patterns		
Nature of Component	Atomic and Immutable Passive		Diffuse and Malleable Active		
Principle of Reuse	Composition		Generation		
Emphasis	Application Component Libraries	Organization & Composition Principles	Language Based Generators	Application Generations	Transformation Systems
Typical Systems	- Libraries of Subroutines	- Obj Oriented - Pipe Archs.	- VHLLs - POLS	- CRT Fmtrs. - File Mgmt.	- Language Transformers

In composition technologies, the components to be reused are largely atomic, and, ideally, are unchanged for their reuse. Examples of such items are code skeletons, subroutines, functions, programs, and objects. Using composition, new programs are derived from building blocks. This software model is the analogous to the hardware activity of plugging together integrated circuit chips to develop hardware systems. In generation technologies, components being reused are often patterns produced by a generator program. The patterns are the seeds from which new, specialized components are grown. Each resulting instance of such a pattern may be highly individualistic, and is more difficult to characterize and isolate.

Reusable patterns have two forms; patterns of code and patterns within transformation rules. In both cases, the effect of the individual reusable

components within the target program tend to be more global and diffuse than the effects of building blocks.

In his two book series, Biggerstaff presents an article by L. Peter Deutsch, titled "Design Reuse and Frameworks in the Smalltalk-80 System," The Smalltalk-80 supports a type of reuse that is unique to the object-oriented approach, that is, reuse of design through frameworks of partially completed code. A framework binds certain choices about state partitioning and control flow. The reuser completes or extends the framework to produce an actual application. Here, the simplest example of a framework is a class that is partially abstract. A class supplies a partial specification and implementation but expects subclasses or parameters to complete the implementation.

[BOE91] Boeing Company, Defense & Space Group, US40 STARS Reuse Concept of Operations, Volume I, Version 0.5, Draft, Informal Technical Data, STARS-SC-03725/001/00, Seattle, WA, August 27, 1991.

In this concept of operations document of 1991, STARS identified functions and processes supporting reuse and organized them into a process framework. As shown in Figure E-1, this process framework has the following four major processes:

1. Reuse Planning
2. Asset Creation
3. Asset Management
4. Asset Utilization

The bulk of the document provides details for all these processes and subprocesses as the STARS Reuse Process Framework.

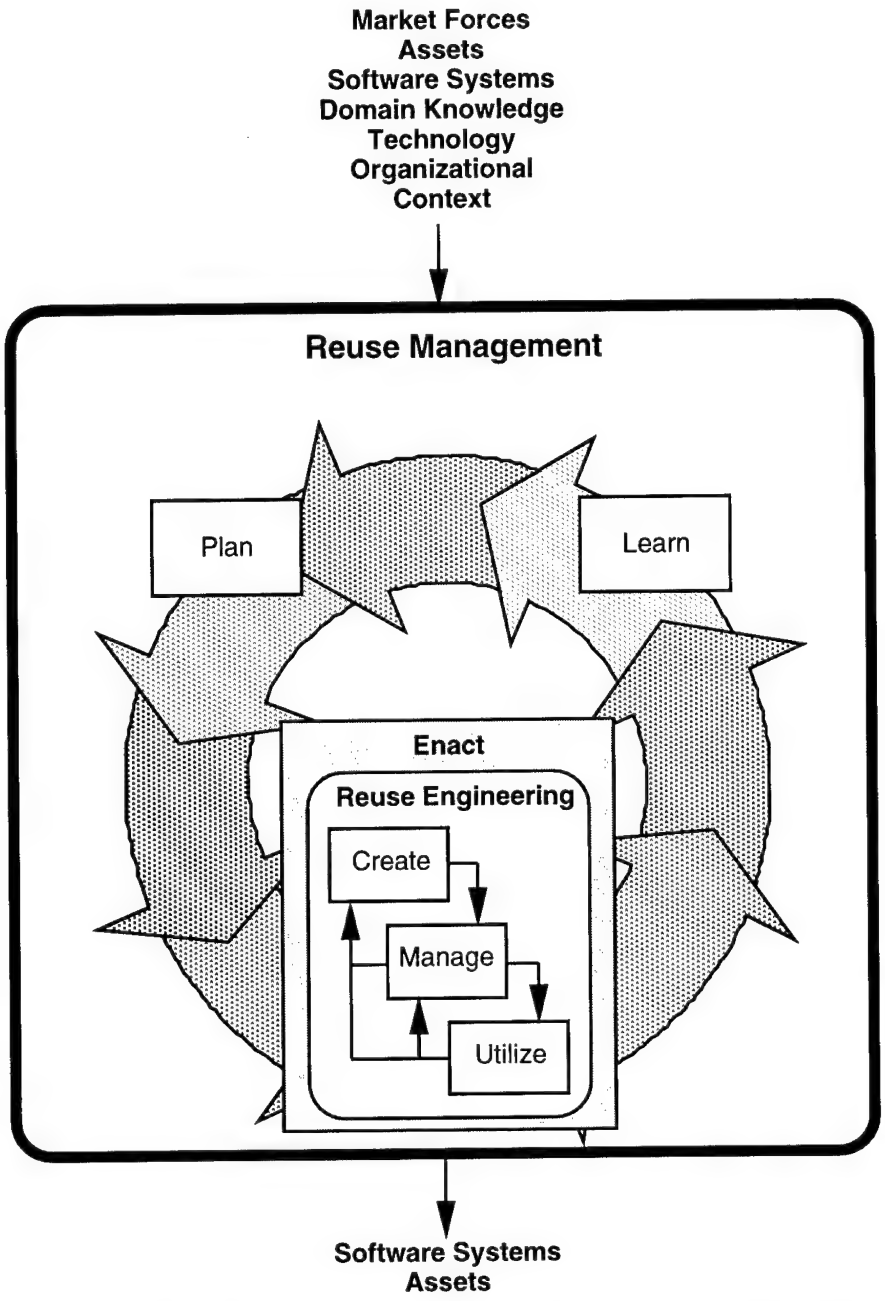


Figure E-1. STARS Conceptual framework for reuse processes

The document concludes with a section about integrating the views of a framework. STARS envisions that reuse in the future will occur in the context of a distributed network of heterogeneous domain-specific libraries. It is likely that each library will focus narrowly on one domain or a small set of vertical or horizontal domains. These libraries are prone to yield high reuse through greater depth of focus and better control of variability. This proliferation of multiple, distributed, domain-specific libraries may be challenging to manage. The STARS document suggests and characterizes a concept for varying degrees of interoperability among these libraries.

[BOE93b] Boeing Company, Defense & Space Group, "STARS Conceptual Framework for Reuse Processes (CFRP)," Volume I: Definition, Version 3.0, STARS-VC-A018/001/00, Seattle, WA, October 25, 1993.

As an update to the 1991 STARS Concept of Operations discussed above in [BOE91], the STARS effort defined a conceptual framework for reuse processes titled the STARS Conceptual Framework for Reuse Processes (CFRP). This document is Volume I of a two volume set; Volume II provides guidance in how to use the CFRP.

The CFRP is a reuse process framework whose scope is limited to identifying the processes involved in reuse and describing, at a high level, how those processes operate and interact. The document is targeted to the Program/Project Planner, the Process Engineer and the Reuse Advocate. The CFRP flows down from the STARS reuse vision and mission. Its authors believe that the CFRP provides a conceptual foundation, a framework, and a set of high level requirements for the reuse technology process and supporting tools needed to accomplish the STARS reuse mission.

These processes are domain-specific in that reusable assets, the development processes, and the supporting technology are appropriate and tailored for a particular application domain. This concept is supported by process-driven engineering, that is, engineering performed in accordance with well-defined repeatable processes that are subject to continuous measurement and improvement and enforced through management policies.

The CFRP consists of dual, interconnected "process idioms" called Reuse Management and Reuse Engineering. The process idioms are further decomposed into process families and these, in turn, are decomposed into process categories. The full decomposition follows.

Reuse Management
- Reuse Planning
  - Assessment
  - Direction Setting
  - Scoping
  - Infrastructure Planning
  - Project Planning
- Reuse Enactment
  - Project Management
  - Infrastructure Implementation
- Reuse Learning
  - Project Observation
  - Project Evaluation
  - Innovation Exploration
  - Enhancement Recommendation

Reuse Engineering
- Asset Creation
  - Domain Analysis and Modeling
  - Domain Architecture and Development
  - Asset Implementation
- Asset Management
  - Library Operation
  - Library Data Modeling
  - Library Usage Support
  - Asset Brokering
  - Asset Acquisition
  - Asset Acceptance
  - Asset Cataloguing
  - Asset Certification
- Asset Utilization
  - Asset Criteria Determination
  - Asset Identification
  - Asset Selection
  - Asset Tailoring
  - Asset Integration

The bulk of the document provides details for all these processes and subprocesses as the STARS Conceptual Reuse Process Framework. Differing from the Boeing report published in 1991, this conceptualization of reuse processes includes a more detailed look at planning and adds enactment and learning as separate families of processes. This document also discusses the linking, recursion and overlapping of these processes.

[CAC95] CACI, Inc. - Federal, "Systems Engineering and Technical Support for DISA/Center for Software," Procedures for Qualification and Engineering of Reusable Assets (Final), U.S. Department of Defense, Defense Information System Agency, Arlington, VA., 1995.

Since the inception of the Software Reuse Program (SRP) in July 1992, DISA has acquired many "lessons learned" which has resulted in re-direction. At the start, the SRP focused on attaining Ada source code for a reuse library, namely the Defense

Software Repository System (DSRS). The original effort to re-engineer assets to make them more generic, and thereby, promote reuse never fully matured as initially thought. Besides being very costly to re-engineer components, the demand for these assets did not justify the time and expense. Users generally wanted to incorporate their own changes to accommodate their own needs. In addition, user feedback indicated that the SRP was not meeting the reusers' real requirements, needs that go beyond Ada source code.

One of the results of this re-direction has been to eliminate certification of assets at Levels 1-4 as reported in [MER93] that follows in this section of the annotated bibliography. This change has cut the costs incurred with dedicated re-engineering, allowing more effort to be spent on acquiring a wider range of reusable assets.

[CAL91] Caldiera, Gianluigi and Victor R. Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer*, February 1991, pp. 61-70.

Caldiera provides a framework of component attributes to help assess its reusability. Figure E-2 shows his "fishbone diagram" that represents the reusability factors and their relationship.

Caldiera associates four metrics to measure these factors and predict the likelihood of reusability. The four selected metrics are defined as follows:

1. Cyclomatic complexity is defined as the cyclomatic number of the control-flow graph of the program.
2. Regularity is defined as the economy of a component's implementation (i.e., the use of correct programming practices).
3. Reuse frequency is the comparison of the number of static calls addressed to a component versus the number of calls addressed to a class of components that are assumed reusable.
4. Volume is based on the way a program uses the programming language.

Each metric has a supporting formula to determine a quantitative value to measure and predict these factors.

In a series of case studies, Caldiera applied his reuse framework using an automated tool and collected metrics. His case studies show that volume, regularity, and reuse-specific frequency have a high degree of independence. Highly reused components have volume and complexity lower than the average, that is, about one fourth of the average. His case studies show that, in general, about only 5-10% of the existing code should be analyzed for possible reuse. Usually, this 5-10% of the code accounts for a large part of a system's functionality.

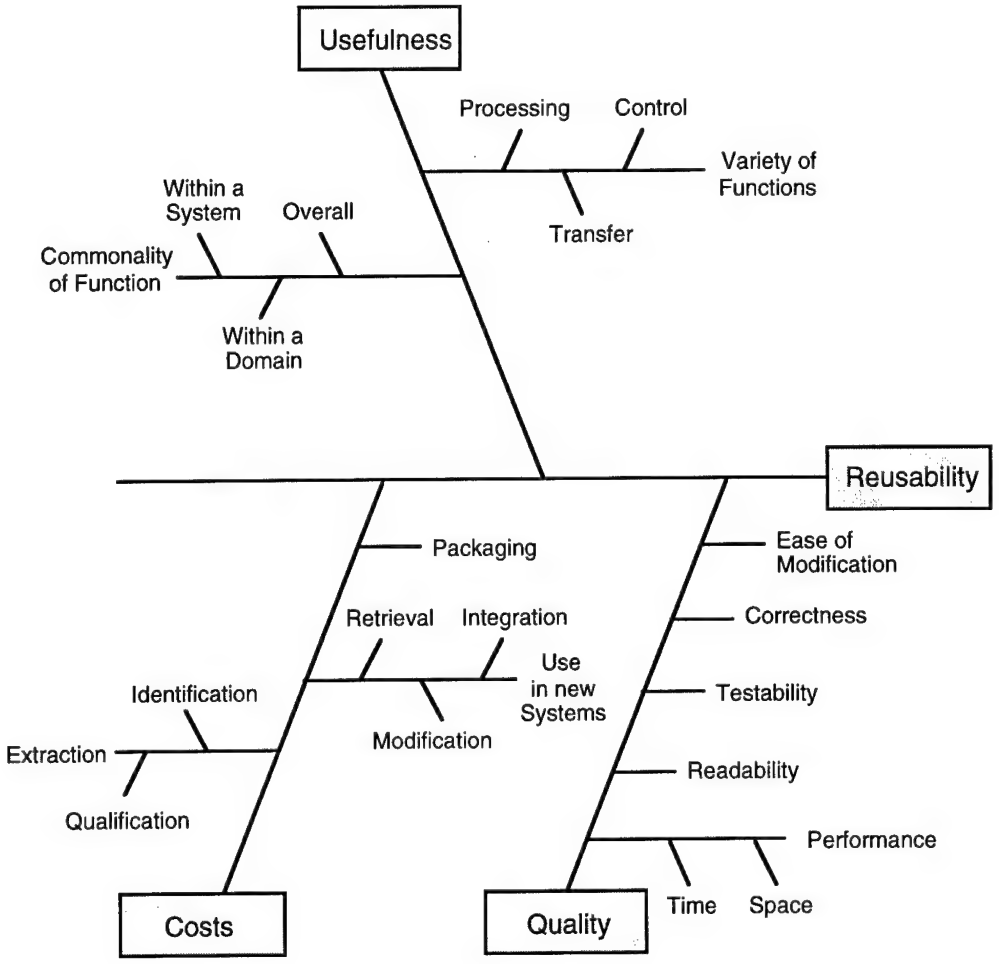


Figure E-2. Factors affecting reusability [CAL91]

[CHU93] Chubin, Sherrie and David Eichmann, David Card, Duane Hybertson, "Software Reuse Program, Software Metrics Plan," Defense Information Systems Agency, Joint Interoperability Engineering Organization, Center for Information Management, DISA/JIEO/CIM, Version 4.1, August 4, 1993.

This Reuse Metrics Plan provides a strategy for identifying, collecting and reporting metrics necessary to assess and improve software reuse processes developed by DISA/JIEO/CIM reuse activities. These activities are collectively known as the Software Reuse Plan (SRP) and is in direct response to the strategy to "define metrics to evaluate success" specified in the DoD Software Reuse Vision and Strategy document of 1992.

Two metrics workshops were held to determine a relevant metrics program. The Goal/Question/Metric paradigm of Basili was used to develop questions that

identified what to measure.[1] The plan identifies four roles in a reuse metrics plan and their relationship to the flow of products and services among them:

1. Repository Manager
2. Program/Project Manager
3. Domain Manager
4. DoD Executive

Details of these roles and activities were identified and a metrics plan for collection and validation was developed in the document. Results can be used in estimation, decision-making and process/product improvement. To ensure that metrics are useful and reliable, each metric is reported quarterly.

[COM95] Comer, Edward R., P1420.1/D5, Guide for Information Technology - Software Reuse - Asset Certification Framework, Technical Committee 4: Asset Evaluation and Certification of the Reuse Library Interoperability Group (RIG) September 1995.

This document describes an Asset Certification Framework that identifies asset certification techniques for a reuse library. An asset certification framework is defined as a technique and associated data model for organizing, selecting, communicating and guiding the process of certifying assets. The certification framework defines a standard interoperability data model for interchanging asset certification information.

The Asset Certification Framework is designed to be a annex to the standard Extended Interoperability Data Model (EIDM) being developed by the Reuse Library Interoperability Group (RIG). The EIDM adds to the P1420.2 Basic Interoperability Data Model (BIDM) by way of its provision for systematic extension. The BIDM defines the minimal set of information about assets that reuse libraries should be able to exchange to support interoperability.

The class hierarchy of the BIDM begins with a RIG object consisting of an asset, an element, a library, and an organization. The BIDM is extended by adding an element type. This element type is related to a certification policy of a library class object.

Four levels of reuse assessment were defined; unassessed, described, analyzed and tested. These levels are used in conjunction with an evaluation process. The software quality evaluation process used in the Asset Certification Framework is derived from the following two standards:

---

[1] Basili, V.R. and H.D. Romback, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, June 1988, pp. 758-773.

1. ISO/EC 9126:1991, Information Technology - Software Product Evaluation and Quality Characteristics and supporting guidelines
2. IEEE P1061, the Standard for Software Quality Metrics Methodology

Using a subset of the certification quality factors of those identified in IEEE P1061, the resulting Asset Certification Framework provides a cohesive structure to derive a metrics methodology and a communication mechanism for management and technical personnel.

The document suggests a set of certification quality factors as follows: completeness, correctness, efficiency, fault tolerance, functionality, maintainability, portability, presentation, reliability, reusability, usability and domain-specific safety, precision and survivability.

[COM96] Comer, Edward R., P1420.1A/D6, Guide for Information Technology - Software Reuse - Asset Certification Framework, Technical Committee 4: Asset Evaluation and Certification of the Reuse Library Interoperability Group (RIG) January 1996.

As an update to the previously annotated reference [COM95] in this section, the September 1995 Guide for Information Technology, Software Reuse and Asset Certification Framework, was provided in January 1996. The differences were slight and are identified below:

1. Descriptions for four categories of reuse assessment were documented.

   Unassessed - characterization of an asset by name only.

   Described - characterization of an asset's meta data, descriptive information about the asset and its intended use.

   Analyzed - certification of an asset's properties using inspection or static analysis methods.

   Tested - certification of an asset's properties or behavior during execution.

2. The detailed data model eliminated the "Quality Factor Type" and was decomposed to only the preceding level of "Certification Quality Factor." This data element contains an description and can include the information about the quality factor type.

These minimal changes from industry review indicate that this work product of the IEEE Standards Project is relatively stable; however, it is still subject to change until its final approval.

[DIS95] Defense Information Systems Agency Center for Software, DoD Software Reuse Initiative, "Software Reuse Business Model (SRBM) Technical Report, January 31, 1995.

This reference, also used in the previous appendices in the categories of Business Strategies and Asset Production defines the area of responsibility called Application Engineering Support. The activities in this area of responsibility consists of operating the library/distributing assets and supporting asset usage. These two activities include the following tasks:

- Operate the library/Distribute assets - consists of managing the library administration, operating and supporting the library and its users, and installing automation to support asset usage
- Support asset usage - consists of applying the asset creator/maintainer's expertise to support asset use on programs, train users in the use of the assets and asset support tools, and support analysis of asset capabilities to meet program requirements

As such, the SRBM is able to integrate many aspects of the reuse context; domains, business strategy, the reuse process, and reuse libraries to make an integrated reuse framework. The SRBM defines a general framework and a particular instance of that framework is possible dependent upon the chosen parameters for each of the variables.

[DOD94a] DoD Software Reuse Initiative (SRI), Technology Roadmap, Version 2.0, Volume 1: Technology Assessment, October 4, 1994.

This Software Reuse Initiative report assesses software reuse technology as not yet matured to the point where a single conceptual framework is accepted by the software community. Several attempts have been made. For example, Biggerstaff and Richter characterized software reuse technologies from the systems involved using a survey of the field. [2] Kruger used a similar perspective with a taxonomy composed of abstraction, selection, specialization and integration. [3] These two spawned a number of approaches that overlapped one another (e.g., compositional reuse, software "backplanes," software "mining," and software repositories). This report provides details on generative approaches, method fusion, model-based, library-based languages and how they related to the Conceptual Framework for Reuse Process.

---

[2] Biggerstaff, T., and C. Richter, "Reusability Framework, Assessment and Directions," *IEEE Software*, Vol. 4, No. 2, pp. 41-49, March 1987.

[3] Fruger, C.W., "Software Reuse," *ACM Computing Surveys*, Vol. 24, No. 2, pp 131-183, June 1992.

Recent approaches in software reuse have focused on process and formalization. Bowles defines the following three dimensions of software reuse: [4]

1. A shift from horizontal to vertical domains
2. A shift from individual to project to enterprise focus
3. A shift from code to design to concept to abstraction

The SRI believes that navigation through these three dimensions outlines an assessment mechanism for the maturity of a reuse capability similar to the SEI's CMM. Capability models for reuse have been proposed in the SPC' Reuse Adoption Guidebook [5] and by the STARS program in 1991 and in 1993. [6,7]

[DOD95a] Department of Defense, "Software Reuse Symposium," March 23, 1995, Huntsville, Alabama.

The Software Reuse Symposium not only provided a forum for new concepts in software reuse, but also provided "tutorial-like" presentations tracing the history of software reuse and evaluating the current state-of-the-art. The following are current program and players in reuse:

Major DoD Reuse Programs

ARPA's Software Technology for Adaptable, Reliable Systems (STARS)

Air Force's Central Archive for Reusable Defense Software (CARDS)

DISA's Software Reuse Program

ARC-Army Reuse Center

Internal Groups

Reuse Executive Steering Committee (RESC)

Management Issues Working Group (MIWG)

Reuse Technical Working Group (RTWG)

---

[4] Bowles, A.J., "The Reality of Software Reuse," *Vista*, New Science Associates, Westport, CT, pp 1-3, May 1993.

[5] Software Productivity Consortium, "Reuse Adoption Guidebook, SPC-920510CMC, Version 01.00.03, Herdon, VA, November 1992.

[6] Software Technology for Adaptable, Reliable System (STARS), "Reuse Library Process Model," IBM STARS Technical Report, CDRL 03041-002, STARS Technology Center, Arlington, VA, July 26, 1991.

[7] STARS, "The Reuse-Oriented Software Evaluation (ROSE) Process Model, Version 0.5, Unisys STARS Technical Report, US-05155/00/00, STARS Technology Center, Arlington, VA, 1993.

External Liaisons

Council of Defense and Space Industry Associations

ACM Special Interest Group on Ada, Reuse Acquisition Action Team (RAAT)

Reuse Library Interoperability Group (RIG)

Industry Reuse Advisory Group (IRAG)

As defined in the DoD Vision and Strategy document of July 1992, software reuse is the application of a reusable software asset to more than one application. Reuse may occur within a system, across similar systems, or in widely different systems. The Vision and Strategy for the Software Reuse Initiative is to move the DoD to constructing software in a way that is supported by process-driven, domain-specific and architecture-centric technologies. The DOD Reuse Strategy has five major thrusts:

1. Implement a product line approach.
2. Develop a reuse-based software system and engineering paradigm.
3. Remove barriers to reuse.
4. Quicken technology transfer.
5. Make successes apparent.

Driven by these thrusts, two volumes of a technology roadmap were published in January 1995.

The emphasis on reuse is increasing because of unprecedented downsizing and movements to reinvent the Government. Defense conversion has included activities to support software reuse such as the Ada mandate of 1991, commercial standards adoption, and best commercial practices and benchmarks. Despite these forces defense software development falls short. The Software Reuse Initiative strongly feels that the remedy is to successfully leverage previously developed assets.

To answer this need, the Conceptual Framework for Reuse Processes (CFRP) was developed by the three STARS prime contractors, MITRE and the SEI. The CFRP is a reuse process framework that provides the following the functions; identifies processes involved in reuse; describes how they might operate and interact; and facilitates managing the transition to reuse. A graphical view of the CFRP and its components are illustrated in Figure E-3.

The CFRP addresses both the management and engineering perspectives. It characterizes reuse in terms of producer-broker-consumer activities and can be used as a checklist in planning a reuse program. It also can be used a way to compare and contrast detailed processes, methods and tools to determine how they meet the needs of an organization and/or project.

Within the CFRP is the Reuse Oriented Software Evolution Model (ROSE). ROSE was developed by Unisys under the STARS program and provides a process framework that bridges the gap between the CFRP and detailed methods. Also part of the CFRP is the Reuse Library Facility (RLF). The RLF is a reuse support tool developed by Unisys and implemented in Ada. RLF emphasizes a structured, domain model-based approach and directly supports the ODM (Organization Domain Modeling), a method for domain modeling. The RLF executes on a SunSPARC station and has a coarse-grained integration to PCTE (Portable Common Tool Environment).

At the Software Reuse Symposium, one of the presentations was given by staff at the Software Engineering Directorate at Fort Monmouth at Army CECOM. This agency maintains that hardware reuse has been successful because of abundant "architectural standards" within systems. The role of a reuse library is to facilitate (not enable) software reuse. The goal is to create software for today's systems that is designed for reuse. For future systems that are developed, components will be deposited and withdrawn from the library. The key to a useful library is the "quality" of the software it contains (i.e., functionality, performance, reliability, architectural compatibility).

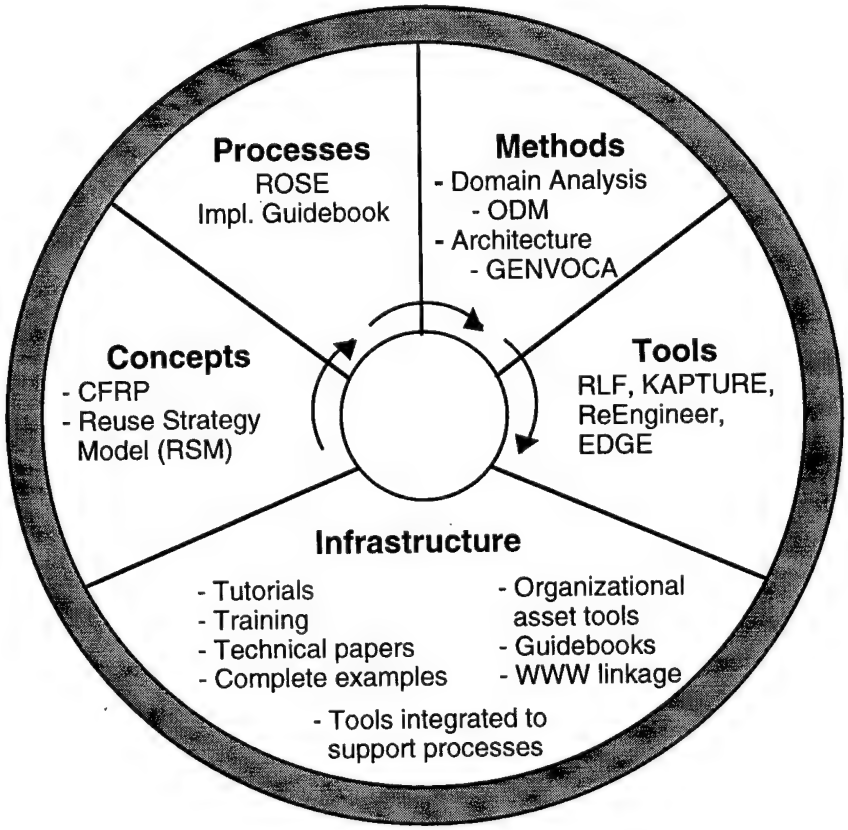


Figure E-3. Conceptual Framework for Reuse Processes [DOD95]

The early emphasis on libraries within the DoD has now shifted to an architectural focus since previously reuse libraries contained software from a variety of sources that has not necessarily been designed for reuse. Current libraries have resulted in numerous deposits and not many withdrawals.

Consequently, developing common templates is a better approach to improve the productivity of the software development process. The emergence of domain engineering helped to establish the concept of an architectural-centric product line.

[DUN92] Dunn, Michael F., John C. Knight, "Certification of Reusable Software Parts," Department of Computer Science, University of Virginia, Charlottesville, VA, and the Software Productivity Consortium (SPC), INF-92-001, August 31, 1992.

Michael Dunn provides a strategy for software component certification and a method to quantify the benefits of reuse. Dunn's approach is based on following premise: Having guaranteed that a specific set of quality guidelines have been adhered to in a set of components, it will then be much easier to verify the quality of a system composed of those components.

Dunn defined three major certification attributes:

1. Life cycle phases
2. Level of granularity
3. Intended domain

Since certification qualities differ for each organization, a framework for them needs to be flexible and accommodate differences.

In addition to techniques and guidelines for certification, the document also established definitions for properties and techniques for domain analysis. Testing definitions, testing guidelines, properties of systems and the economics of certification were established. Two case studies were presented that applied the techniques and guidelines. Dunn's concept was intended for member companies of the Software Productivity Consortium (SPC).

[FRA92] Frakes, William, Ruben Prieto-Diaz and Edward Comer, "Ada Software Reuse and Domain Analysis Seminar, presented at the Clarion Plaza Hotel, Orlando, FL, November 16, 1992.

In his presentation at this Orlando seminar, Bill Frakes defined a reuse maturity model as shown in Table E-2. Frakes' reuse maturity model is an enhancement of that which Koltun and Hudson developed in 1991. Frakes uses this

framework to assess reuse on a continuum of maturity levels from initial and chaotic to ingrained, and across several business and technical dimensions.

Table E-2. Reuse maturity model [FRA92]

	Initial, Chaotic	Monitored	Coordinated	Planned Ruse	Ingrained
Motivation, Culture	Reuse discouraged	Reuse encouraged	Reuse incentivized, re-enforced, rewarded	Reuse indoctrinated	Reuse is the way we do business
Planning for reuse	None	Grassroots activity	Targets of opportunity	Business imperative	Part of strategic plan
Breadth of reuse	Individual	Work group	Department	Division	Enterprise wide
Responsibility for making reuse happen	Individual initiative	Shared initiative	Dedicated individual	Dedicated group	Corporate group (for visibility not control) with division liaisons
Process by which reuse is leveraged	Reuse process chaotic, unclear how reuse comes in	Reuse questions raised at design reviews (after the fact)	Design emphasis is placed on off-the-shelf parts	Focus on developing families of products	All software products genericized for future reuse
Reuse Assets	Salvage yard (no apparent structure to collection)	Catalog identifies language and platform-specific parts	Catalog organized along application-specific lines	Catalog includes generic data processing functions	Planned activity to acquire or develop missing pieces in catalog
Classification activity	Informal, on an individual basis ("in the head," or "in the desk")	Multiple independent schemes for classifying parts	Single scheme, catalog published periodically	Some domain analyses done to determine categories	Formal, complete, consistent, timely classification
Technology Support	Personal tools, if any	Many tools (e.g., CM), but not specialized for reuse	Classification aids and synthesis aids	Electronic library separate from development environment	Automated support integrated with development environment
Metrics	No metrics on reuse level, payoff, or costs	Number of lines of code used in cost models	Manual tracking of reuse, occurrences of catalog parts	Analyses done to identify expected payoffs from developing reusable parts	All system utilities, software tools and accounting mechanism are instrumented to track reuse
Legal, Contractual, Accounting considerations	Inhibitor to getting started	Internal accounting scheme for sharing costs and allocating benefits	Data rights and compensation issues resolved with customer	Royalty scheme for all supplies and customers	Software treated as a key, capital asset

[MER93] Merritt, Steven, "Framework for Certification of Reusable Software Components," DISA/CIM Software Reuse Program, February 26, 1993.

This document provides guidelines for the certification of reusable software components. Specifically, the following multiple levels of certification are recommended as a practical way of providing a rating of components within a reuse library:

> Level 1 certification identifies the component as approved for installation.

Level 2 certification identifies the component as released to users and verified for completeness (i.e., source code must compile).

Level 3 certification identifies the component as tested with test data and test results captured and available.

Level 4 certification documents a reuser's manual for the component which is available for distribution.

The requirements of each level subsumes the requirements of the previous level. In additional to the defined levels, a detailed process for certifying reusable software components for installation into a reuse library was modeled using the IDEF method.

[MOR89] Moore, John A. and Sidney C. Bailin, "Domain Analysis: Framework for Reuse," Technical Report, Computer Technology Associates, Rockville, MD, October 1989.

John Moore proposes a life cycle approach to domain analysis and reuse-based software development. He believes that domain analysis is complementary and parallels the on-going process of system development. Moore also believes that reuse-based development relies on the economics of supply and demand. The developers supply reusable resources which includes their domain analysis and other associated reusable products.

[PAY88] Payton, Teri F., "Reusability Library Framework," Presentation at STARS Foundations Workshop, Unisys Defense Systems, Paoli, PA, April 1988.

Unisys' Reusability Library Framework (RLF) project under STARS was intended to provide a general framework and a set of tools to support the creation and maintenance of a repository of reusable Ada software components. The RLF is organized around application domains. Unisys believes the most effective gains in productivity will be from using libraries of components from specific domains during software development.

[SOL89] Solderitsch, James J. , Kurt C. Wallnaw and John A. Thalhamer, "Constructing Domain-Specific Ada Reuse Libraries," *Proceedings of the Seventh Annual Conference on Ada Technology,* U.S. Army CECOM, Ft. Monmouth, N.J., March 1989, pp. 419-433.

James Solderitsch believes that high impact reuse is achieved by focusing on specific application domains as does the RLF. RLF supports domain modeling and repository management. The domain modeling consists of knowledge representation components interfaced to the library with varying functionality and points of view. The repository management includes insertion, classification, qualification, and retrieval of components.

Appendix F - Technical Paper

This appendix consists of a technical paper titled "Certification of Reusable Software Components." SPS submitted this paper for juried review to the Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), held jointly with the 6th Complex System Engineering Synthesis and Assessment Technology Workshop (CSESAW '96) and the 4th IEEE Workshop on Real-Time Applications (RTAW'96) on October 21-25, 1996, in Montreal, Quebec, Canada.

The goal of this conference is to bring together industrial, academic, and Government experts from various disciplines, to determine how the disciplines' problems and solution techniques interact with the whole system. Researchers, practitioners, tool developers and users, and technology transition experts will participate. Tracks are planned in the following areas:

- AI and Intelligent Systems
- Architecture, Tools, Environments, and Languages
- Database and Data Management
- Dependable Real-Time Systems
- Formal Methods
- Heterogeneous Computing
- Software Engineering, Re-engineering, Reuse
- Standards
- Systems Engineering
- Virtual Reality, Multimedia, Team-Time Imaging

This paper was submitted for the track discussing software engineering, re-engineering and reuse. In this track as well as others, long-term research, near-term complex system requirements and promising tools, and existing complex systems and commercially available tools will be examined on a level playing field.

Certification of Reusable Software Components

Summary of Work In Progress

Sharon L. Rohde , Karen A. Dyson, and Pamela T. Geriner, Ph.D.
Software Productivity Solutions, Inc., Indialantic, FL, USA
Deborah A. Cerino, USAF Rome Laboratory, Rome, NY, USA

Abstract. *This technical paper provides a synopsis of in-progress research and development in reuse and certification of software components at Rome Laboratory of the Air Force Materiel Command, Rome, NY. A Certification Framework for software components has been developed which is sensitive to varying domains, business strategies and asset types. A cost benefit plan, an operational concept, a suite of certification tools, and a prototype have been defined. Field trial procedures have been developed, initially applied, and the results reported. Additional field trials are planned for Maxwell Air Force Base, Gunter Annex, Alabama, and Underwriters' Laboratory, Research Triangle Park, NC.*

Motivations. It has been estimated that the U.S. Department of Defense (DoD) spends in excess of $24 billion per year to develop and maintain software for weapons, command and control, and other automated information systems [1]. The increase of software intensive systems in conjunction with rising software development and maintenance costs has resulted in the need to identify methods that will accelerate development schedules, lower cost, and improve quality. To address this problem, the DoD established a program in November 1991, for implementing initiatives in software and other information technologies. As part of this program, the Director for Defense Information proposed a software reuse initiative to build partnerships among users and suppliers of reusable components as well as the research and development community. Developing certification standards for components was one of the key elements of the DoD's software reuse strategy.

The DoD's software technology strategy also states that the savings from reusing software assets is estimated to be $11.3 billion in constant 1992 dollars by the year 2008. The General Accounting Office reports that "benefits go beyond cost saving to include substantial increases in productivity for avoidance of rework, and added software quality through the use of tested components [1]." Recognizing that software will not be reused unless its quality can be accurately and effectively determined, Rome Laboratory (RL) of the United States Air Force Materiel Command, Rome, NY, established a research program in reusable software asset certification. Certification is expected to increase the reduction of software costs by stimulating component reuse and reducing the amount of rework required [2].

Project. In January of 1994, RL began an exploratory development effort entitled "Certification of Reusable Software Components" (CRC). The goal of this technology thrust at RL is to make certification usable, practical and cost-effective. CRC has the following project objectives:

- Select only the practical, usable, and cost-effective subset of reliability and quality techniques that improves the confidence in reusable software.
- Synthesize these techniques into a cohesive framework that is sensitive to different user certification requirements.
- Make certification understandable, practical and usable for the typical engineer by hiding the theories and complexities.
- Design a cost-effective certification process in terms of a return on investment with quantified costs and benefits.
- Refine and demonstrate a piece of a Certification Framework that is usable, pragmatic, and cost-effective for near-term application in Government, contractor and commercial reuse environments.

RL's 20-year legacy of research in software quality, measurement, test and verification provides an excellent foundation for certification research and development.

The prime contractor for CRC is Software Productivity Solutions, Inc., Indialantic, FL, with subcontractors from General Research Corporation, Santa Barbara, CA and VeriQuest,

LLC, Raleigh, NC. The distinguished Project Review Team consists of Ms. Deborah Cerino, RL's Project Manager and representatives from Underwriters' Laboratory, Raleigh, NC, and MITRE, McLean, VA.

With the downsizing and reorganization within the Government, the pressure is on to demonstrate transferable, usable technologies. To facilitate transfer of certification technology, RL is initiating a Memorandum of Agreement (MOA) with the Gunter Annex of the Maxwell Air Force Base, AL, through Ms. Judy Roberts, Program Manager of the Air Force's Reuse Center (RC). Gunter is planned as a beta test site for trial use of the certification technology developed under CRC. Gunter's planned participation will validate the underlying ideas while providing valuable information for enhancement, refinement and continued exploration.

Measures of Success. Early in the CRC project, three measures of success were defined as appropriate for this research and development effort. Figure F-1 illustrates how CRC's measures of success span three areas: Innovation, Experimentation and Validation. The innovation "wedge" of this upwardly progressing arrow is our theoretical development of new certification concepts. Experimentation, the second wedge, is the application of these theoretical developments in a laboratory environment. Validation is achieved through analysis of the results of applying our theories to real-world situations. A preliminary project assessment using each of these three measures concludes this paper.

Definitions. Traditionally, the term *certification* has been used to refer to a process whereby an independent organization confirms that products meet certain requirements [3]. Within the software reuse community, the term refers to a variety of activities including inspection and documentation of reusable assets as well as quality evaluation and assessment. For CRC, *certification* refers to a process in which inspection, analysis, and testing techniques are used to achieve assurance of the quality of reusable assets. This certification process may be performed by a reuse repository, by a reuser, by an independent organization providing such services, or by a development organization.

State of the Practice. The Defense Information Systems Agency (DISA) surveyed a group of repository personnel and experts on reuse [4]. DISA found that there was very little empirical data on either code asset or non-code asset reuse. In addition, their findings indicated that 70% of the respondents agreed that reuse certification is a necessary activity, 45% were not aware of any standards or do not use standards in their certification, and 90% did not know the actual cost of certification. Consequently, this DISA study recommended that cost and benefits of certification be evaluated in order to provide the DoD advice on future resource allocations for certification activities.

Several other studies have assessed the state-of-the-practice of certification for reuse. Development reuse organizations "wanted the Government to 'certify' the testing level that a component has undergone and the reliability of the component so that a contractor does not have to duplicate similar testing procedures [5]."

One area that requires more research is certification criteria. Current reuse programs have indicated that their main concerns are centered around the criteria of completeness, correctness, understandability, and modularity [6]. Overall, the conclusion of studies such as these is that the state-of-the-practice of certification is still immature and further research is needed in certification processes, techniques, and tools in order to identify the cost-effective approaches.

Reuse Context. To bound the scope of the proposed work for this project, we constructed a diagram of the reuse context to determine our realm of operation. The reuse context is the set of circumstances and requirements within which reuse is carried out. Since certification is a part of the overall reuse process, it was necessary to determine which elements of the context affect certification and which ones are affected by certification. Figure F-2 illustrates our context, the Reuse Context for Asset Quality Certification.

This reuse context consists of several elements:

- Business Strategies
- Domain
- Asset Type
- Reuse Framework
- Reuse Library
- Asset Production
- Asset Selection
- Asset Certification

The elements of Business Strategies, Domain, and Asset Type drive the Frameworks for both reuse and certification. The Domain refers to either the application domain for which an asset is developed or the application domain in which an asset will be reused. To determine the

particular domain for a reuse context, a domain analysis should be performed.

Asset Production, Asset Selection and Asset Certification are basic reuse activities within the Reuse Process, and all may be performed using an associated Reuse Library. Asset Production is the process by which an asset is developed and made ready for inclusion in a library or repository. Asset Selection is the process by which a potential reuser searches the library and selects candidate assets for use in a new system. Asset Certification is the process by which an asset is evaluated for conformance to the requirements it must satisfy to be reused. These activities of the Reuse Process are driven by the Reuse Framework through guidelines, standards, classification techniques and measurement.

Within the reuse context, the focus of CRC is upon a subset of the Reuse Framework called the Certification Framework (CF) as well as its algorithm for asset certification. Very little work has been done in these areas, yet both are influenced by the other elements of the reuse context. Using this reuse context, we documented a literature survey of research in the areas of domain analysis, business strategies, asset production, asset selection and reuse frameworks [7]. This survey provided the technical foundation for our effort and its results impacted our development of the CF.

Certification Framework. The purpose of the CF is three-fold:

1. Define the elements of the reuse context that are important to certification.
2. Define the underlying models and methods of certification.
3. Define a robust, decision-support technique to construct a context-sensitive process for selecting the techniques and tools and applying them in order to certify assets [8].

A tabular view of the CF is shown in Table 1 and vertically lists the elements that compose it; the software reuse business model [9] [10], the domain [11], asset type, quality factor [12] [13] [14], non-conformance class, certification techniques, and the certification process. The CF provides a broad view of the reuse context. We are conducting a series of field trials based upon particular threads through the tabular view of the CF, using specific attributes in each of the vertical elements.

Cost/Benefit Plan. During the CRC effort, we also developed a Cost/Benefit Plan that describes a systematic approach to evaluating the costs and benefits of applying certification technology within a reuse program [15]. The plan focuses on the benefits of certification in terms of risk reduction; it quantifies the risk reduction effect in terms of cost avoidance. The plan includes a synopsis of general reuse cost/benefit models as additional supporting materials.

The cost model for certification is based on the type of error (or defect) and the ability to detect its presence as it impacts the cost of rework. Since resources usually prohibit exercising all possible defect detection techniques, the model determines the order in which methods should be applied in a certification process to maximize benefit, in terms of reduced risk or rework due to defects.

Our approach maximizes rework avoidance with respect to a technique's defect detection effectiveness, investment cost, and incremental cost. As identified in equation (M-5), our stepwise certification cost effectiveness algorithm is used not only to calculate the costs and benefits associated with defect detection methods, but also the order in which the methods are applied. The result is a certification algorithm that can be optimized for a specific organization's requirements.

Since most organizations may not have all the information available to exercise our certification algorithm, we defined a default profile, based on empirical data collected from studies of industry projects. Our default profile can be used to "get started" and can be fine-tuned with organizational data, as it becomes available. Our default profile is optimized for the quality factor of "Correctness" and the asset type of "Code." If organizations are interested in other quality factors, the CF provides guidance on the selection of other techniques and tools.

Process. Given the CRC CF and the Cost/Benefit Plan, we constructed a generic, context-sensitive default certification process, as shown in Figure F-3. The default certification process consists of four main steps: Readiness Assessment, Static Analysis, Code Inspection and Testing. The default process certifies code components (as opposed to other types of reusable assets) and addresses the certification concerns of Completeness, Correctness and Understandability. We developed detailed procedures, data collection forms and guidelines to support the successful execution of the default certification process in our field trials.

Tools. Prior to conducting the field trials, we developed a method for selecting tools based on certification tool requirements. Using this method, we derived a "best bet" list of candidate tools that could be effectively used for

certification [16]. This list of candidate tools represented the best value in terms of functionality, ease of use, price, performance and integration. The recommended tools are intended for a non-developmental certification organization; that is, one that has no existing tools and does not actively develop software. The recommended tool list would be quite different if the organization had existing tools and was actively engaged in software development.

From this "best bet" list, the following tool environment was selected for the initial field trial:

- AdaWise - provides static analysis of alias usage, elaboration order and order dependencies
- Logiscope - provides static and dynamic analysis of control flow diagrams, and structural testing support
- AdaQuest - provides static analysis of style guidelines, size and complexity

The Rational APEX environment supplied the compiler, debugger and code manager while executing on a Sun SPARCstation with the Solaris 2.4 operating system.

Field Trials. The field trials are a "hands-on" test of the default certification process as applied to an asset. Results of the trials help assess the accuracy and understandability of the procedures to conduct certification, the effort required to collect data, and the effectiveness of techniques in detecting defects in assets.

We conducted an initial field trial by selecting an asset to certify, sized at a two staff-week effort (i.e., employing one Certification Analyst and one Certification Engineer). We selected asset #157, the ProGen utility, from the ASSET (Asset Source for Software Engineering Technology) repository distributed on the Walnut Creek Ada CD-ROM.

The ProGen asset is 1,543 logical lines of code (Ada semicolons), or 4,387 physical lines of code (non-blank lines). It consists of 10 Ada packages. The asset was large enough to not be trivial, and small enough to be certified within a two staff-week effort. ProGen is a utility program that automatically generates prologues for Ada code files. It extracts information such as pragmas, types and representation clauses to construct a prologue. The asset includes a main procedure to generate a single executable. It had no recorded defect history.

The Certification Analyst reviewed the ProGen source code by desk-checking and found 2 major defects and 11 minor defects.[1] Therefore, the Certification Analyst seeded 5 additional major defects into the asset to provide a significant number of major defects known to her in advance of the field trial. No minor defects were added. The seeded defects were not created in an attempt to duplicate a particular defect profile (i.e., distribution of defect types). The known defects were not shown to the Certification Engineer prior to conducting his tests.

Results. The results of the initial field trial and its data analysis reported here describe the defect detection, the asset's defect profile, and the effectiveness of the techniques used. Additional details regarding secondary findings are available [17].

Many more natural defects were found in the asset during the field trial than were known prior to the start. Figure F-4 shows how many defects were found versus how many are known to exist at completion of the field trial. Defects are counted as unique defect reports (i.e., if the same defect was detected by more than one technique, it was counted only once). Defects categorized as *not found* must be, by definition, either seeded defects or those found by the Certification Analyst during her desk-check code review.

Figure F-5 indicates the defect profile of the asset in terms of the known defects. Note that there are seven uncategorized defects that were found during testing. It is important to understand that defects reported during testing are actually failures, and it is not until a failure is debugged that it can be attributed to specific units and lines of code. Debugging was not done as part of the field trial.

As shown in Table 2, the defect density of the asset's major defects, including the seeded defects, is about average for Ada code [15]. Major defects, as defined for the field trials, are equivalent to what are typically reported as defects (or errors) in industry. The number of minor defects was surprising; however, most of these were style guideline violations.

Figure F-6 compares the asset's defect profile, including both major and minor, seeded and natural defects, to the default profile. One notable difference is that there is a much lower proportion of computational defects. A likely explanation for this difference is that this particular asset is not computational in nature.

[1] A major defect is defined as an error that prevents completion of a certification step or results in a failure during testing, whereas a minor defect may be non-conformance to a style guideline.

This finding indicates that we cannot assess the effectiveness of our default process in detecting computational defects based on this initial field trial.

In certification, it will typically be the case that an individual asset defect profile is different from the default profile of any given group of assets. The more that is known about the expected defect profile of assets to be certified, the more cost-effective a process can be designed to certify them. For example, if a group of assets to be certified is known not to be computational, then one would not need to include a technique that is effective at detecting computational defects.

Another aspect of technique effectiveness can be derived from Figure F-4, previously described. All but one of the known major defects was found, and the one not found was a seeded defect. Effectiveness of the default certification process at finding defects is better represented by the proportion of the total seeded defects found than by the proportion of known defects found. This is because there are probably additional natural major defects in the asset, so the total number of defects in the asset is unknown. Therefore, this field trial resulted in 80% effectiveness at detecting seeded defects, as shown in Table 3.

Figure F-7 indicates the cumulative effectiveness of the steps in the certification process where effectiveness is defined as the proportion of known defects found. From this we can draw several important conclusions. We cannot, however, claim that the combined effectiveness of the default certification process is 90%. As discussed previously, we do not know the total number of defects in the asset. Furthermore, based on the effectiveness at finding seeded defects, we have reason to believe that more natural defects exist.

Again looking at Figure F-7 and the first certification step, Readiness Assessment, there were no defects found. This result indicates that all code needed to create an executable was available and compiled without error. Readiness Assessment was intended to address the certification concern of Completeness.

As for the Static Analysis step, only minor defects, and no major defects, were found in the asset. However, the 55% effectiveness rating shown on the graph may be misleading. The automated tools used in this step are virtually 100% effective at finding the defects that they are designed to find. The effectiveness rating indicates that what the tools are designed to find were only about half of the known minor defects in the asset.

The Code Inspection step found only one-third of the major errors. This was disappointing, and likely explanations are as follows:

- Highly effective inspections reported in the literature are usually multi-person techniques and our certification process used a single inspector technique (i.e., the Certification Engineer).
- Our checklist approach for data collection may focus too much attention on the checklist at the expense of a deeper understanding of the code.
- The inspection technique may be weak at finding logic defects.

And lastly, the Testing step, excluding the cumulative effects of techniques from other steps, found two-thirds of the major defects. This value can be calculated by subtracting the effectiveness of the Code Inspection step from that of the Testing step. All defects found during the Testing step were, by definition, considered major defects.

In terms of certification, the asset failed in two certification concerns, that is, Correctness, and Understandability; the asset passed Completeness. In practice, the Certification Engineer would face the following choices with a failure:

- Reject the asset.
- Report the assert as uncertified and record all known defects.
- Return the asset to the donor and request repair of known defects; repeat the certification process after repairs.
- Repair the defects; repeat the certification process after repairs.

Some certifiers may choose to include defect repair as part of their certification process and recertify after repairs have been effected, depending upon the nature and the number of the defects found. Repeating the certification process ensures that the defects were repaired and detects any new defects inserted as a result of the repair activity.

Lessons Learned. An unexpected lesson learned from this initial field trial was that combining all of the steps in our certification process was highly effective in detecting defects. Each certification step tended to find different types of defects. For example, Figure F-7 shows that all of the major defects would have been missed if we had performed only Readiness Assessment and Static Analysis. The results also indicate that we would not have wanted to jump into the Testing step without having performed the preceding three steps.

Because of this finding, we now believe that a "single-technique-per-step" certification policy, which is typical of many existing reuse repositories, may not make sense. Instead, we believe that techniques should be applied and evaluated, in combination.

Our initial field trial provided valuable data for future experimentation, as well as validation and refinement of the certification process, the tools, and techniques. These refinements will be applied to the succeeding field trials planned for Gunter and UL.

Prototype. A computer-based prototype that automates many of the aspects of the CF is under development and is planned for delivery to RL in June 1996 at the end of the CRC contract. The prototype can be accessed through the World Wide Web through a CRC home page and demonstrates features of the CF.

Conclusion. Much has been accomplished under the CRC effort to date:

- An assessment of the state-of-the-practice for reuse and certification and their supporting technologies.
- A Certification Framework that is adaptable to a wide variety of domains, business strategies and asset types.
- A Cost /Benefit Plan that uses probable rework as a measure of risk and shows the economic value of certification in a reuse program.
- A certification cost model that is tailorable to an organization's requirements and provides a method to tradeoff certification benefits and costs.
- A certification algorithm that defines the processes and tasks to isolate and analyze defects by type and severity.
- An evaluation of static analysis and testing techniques that can be used to create a certification environment that is site-specific.
- Results from initial field trials and detailed procedures and guidelines to perform succeeding field trials at different sites.

Using the measures of success previously defined, a preliminary assessment of our project accomplishments is positive. Examples of our innovations in theoretical developments include the CF itself, the Cost/Benefit Plan and its associated work products (i.e., the code defect model and the cost/benefit model). Experimentation was achieved by initial field trials and other supporting items to execute the field trials (i.e., field trial procedures and guidelines, tools evaluations, user interviews and operational concepts). Validation is planned through application of our innovations to real-world situations at Gunter and UL.

All of these achievements and their lessons learned, however, should be viewed within the context of the phases and milestones of technology maturation as illustrated in Figure F-8 [18]. Redwine found that the average timeframe for a technology to mature from Milestone 0 to Milestone 4 is approximately 15-20 years. Widespread use can take another decade. The technology of certifying reusable software components was clearly in the Basic Research Phase prior to Milestone 0 at the initiation of the CRC project, even though some supporting reuse technologies are more mature (e.g., domain analysis, asset production, asset selection, and reuse libraries). The products of CRC have helped advance certification technology into the phases of Concept Formulation and Development and Extension. Within this roadmap to technology maturation, a plan for certification is feasible. Moving toward the Popularization Phase and beyond is achievable, but will require considerable time and effort.

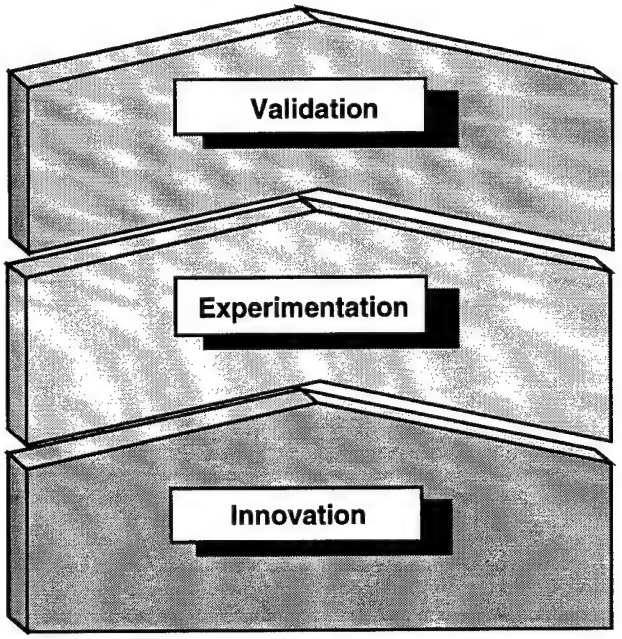


Figure F-1. Measures of success for CRC

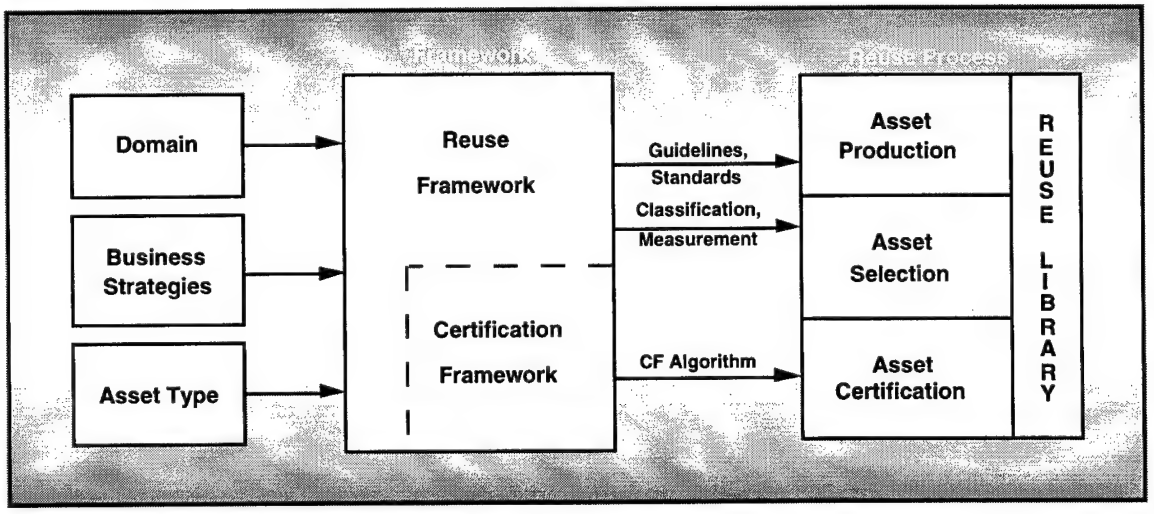


Figure F-2. Reuse Context for Asset Quality Certification

Table F-1. Tabular view of the Certification Framework

S/W Reuse Business Model	Domain	Asset Type	Quality Factor	Non-Conformance	Techniques	Certification Process
Vendor Owned Domain	MIS	Design Info	Correctness	Latent	Compilation	Process Definition
Gov't Supported Standard	Avionics	Document	Completeness	Robustness	Static Analysis	Procedures
Value-Added Reseller	C^2	Test Artifacts	Understandability	Validation	Inspection	Tools
Gov't Owned Architecture	Automated Test Equip.	Req. Specs	Performance	Interoperability	Testing	Data Collection
Gov't Owned Domain	Weapon Systems	Code	Fault Tolerance	Operational	Formal Verification	Certification Levels
Reengineering	Communication	Architecture	Functionality	•	Benchmarking	
Public Library	Intelligence	Database Schema	Maintainability	•	Modeling	
Commercial Library	Process Controls	Models	Portability	Other	•	
	•	Video	Reliability		•	
	•	•	Usability		•	
	•	•	Safety		Other	
	Other	•	Security			
		Other	Availability			
			Testability			
			Survivability			

Equation **(M-5)**

$$\max \; Ca_k = \sum_1^m \; \sum_1^n (D_i \bullet RH_i \bullet LR) \bullet DD_{ij} - \sum_1^m (Inv_j + Inc_j)$$

$$\text{w.r.t} \qquad \sum_1^m (Inv_j + Inc_j) \le B$$

where

Ca_k = cost avoidance due to certification of asset k
D_i = defect density for defect type i
RH_i = number of rework hours for defect type i
LR = hourly labor rate
DD_{ij} = percent of defect type i detectable by technique j
Inv_j = investment cost for technique j
Inc_i = incremental cost for applying technique j
n = number of defect categories
m = number of certification techniques
B = budget for certification activities

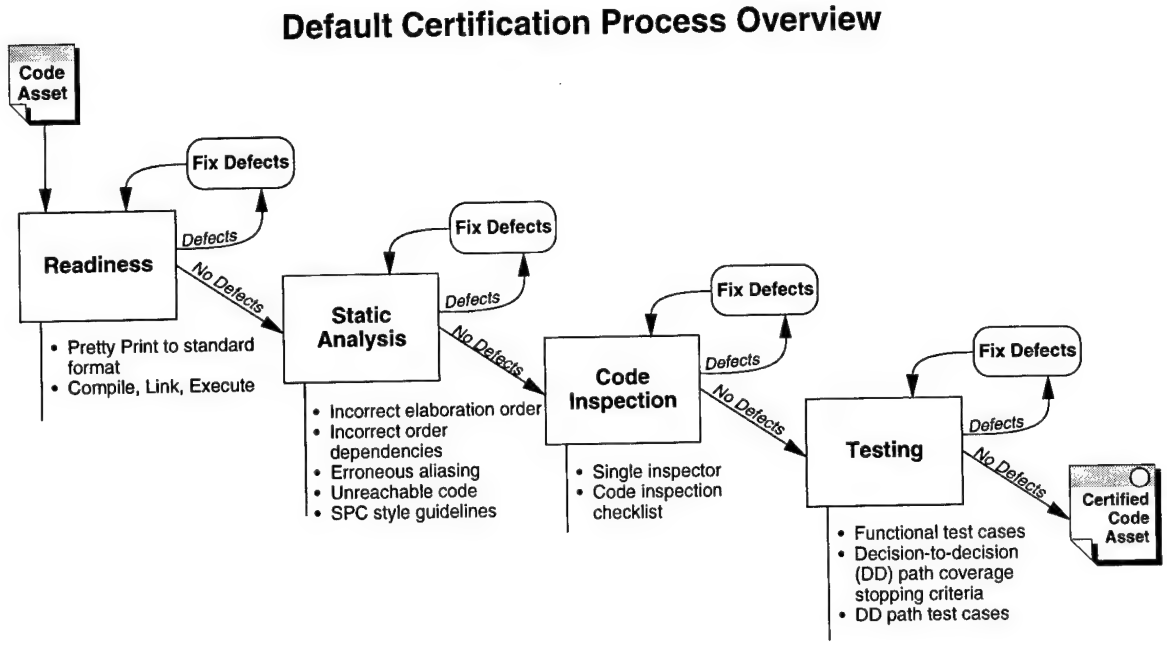


Figure F-3. Default certification process overview

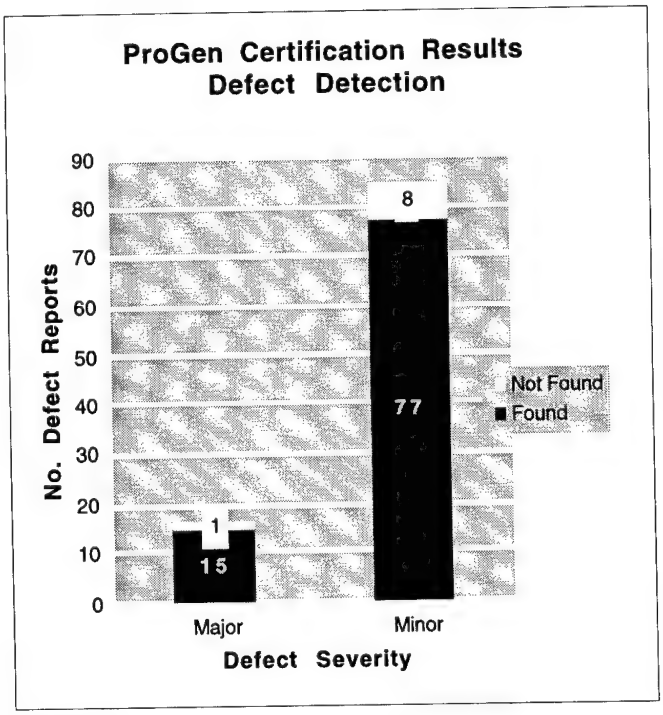


Figure F-4. ProGen certification results of defect detection

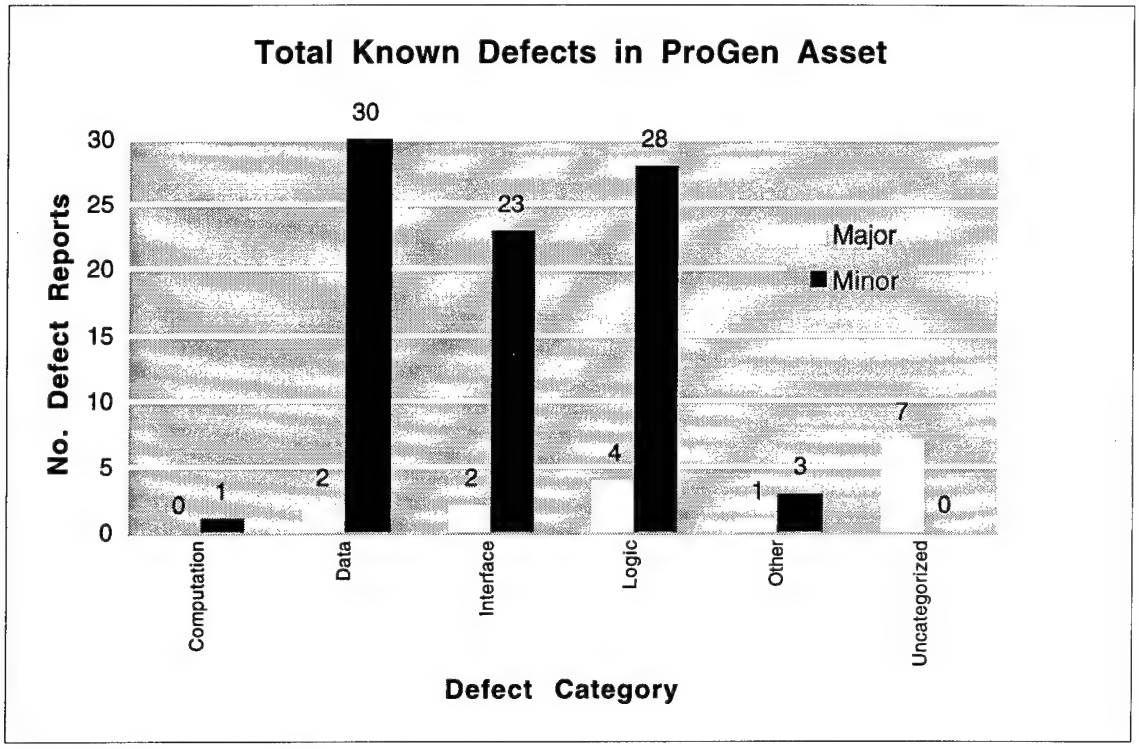


Figure F-5. Asset's defect profile

Defect Density

Defect Severity	Defect Density (defects/1000 physical lines)	
	Asset's	Average for Ada
Major	4	5
Minor	19	Data not reported

Table F-2. Asset's defect density

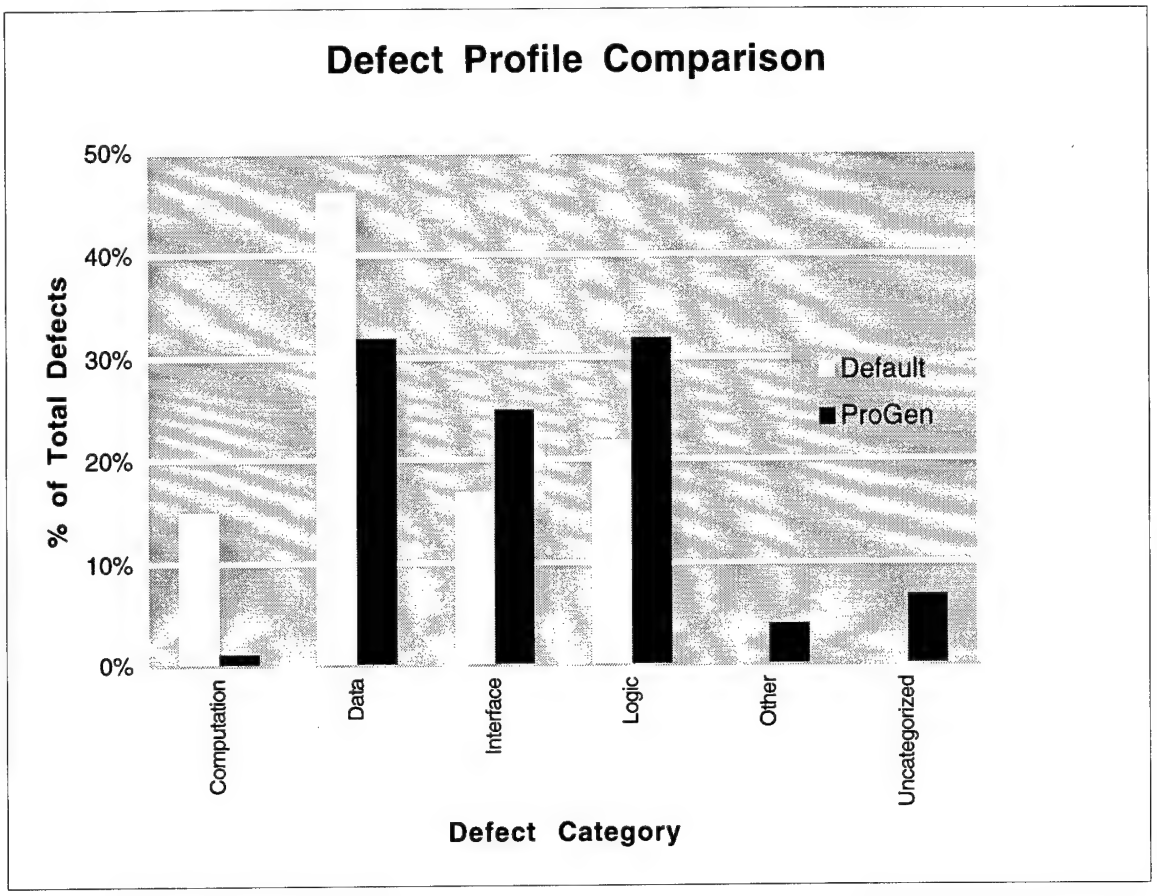


Figure F-6. Comparison of asset's defect profile to default profile

Table F-3. Effectiveness at detecting seeded defects

Found	Known	Effectiveness
4	5	80%

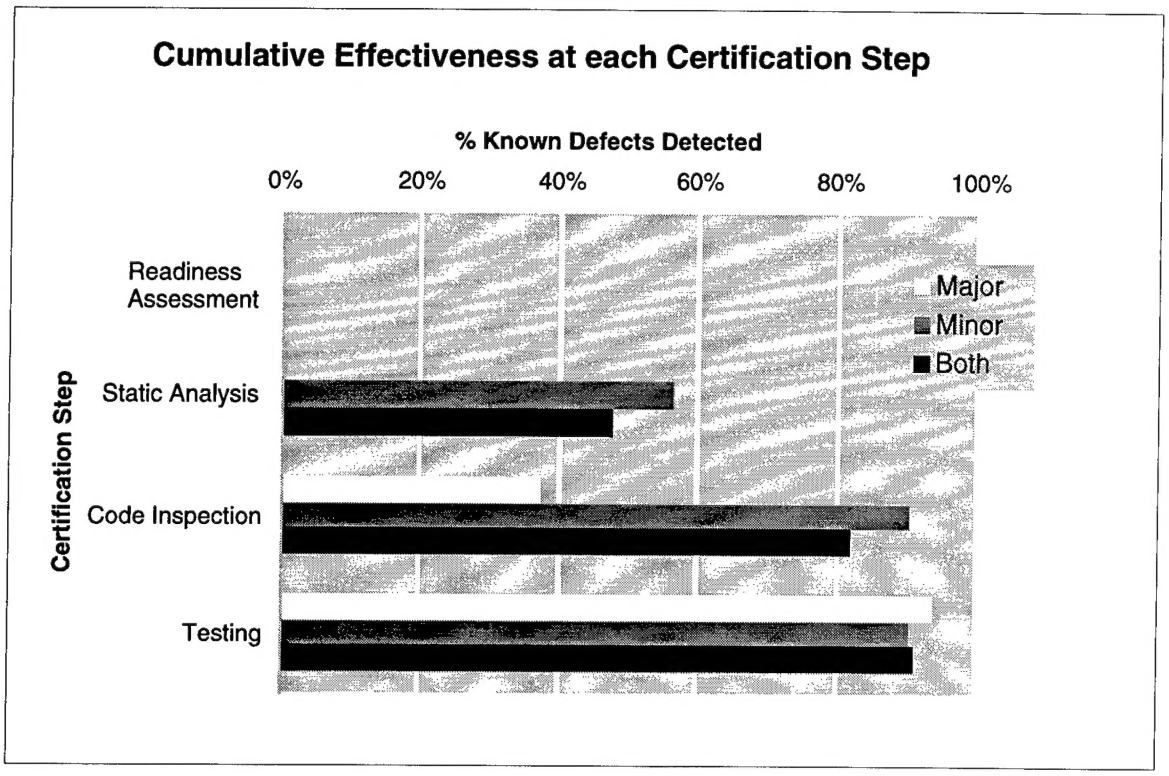


Figure F-7. Cumulative effectiveness of certification steps

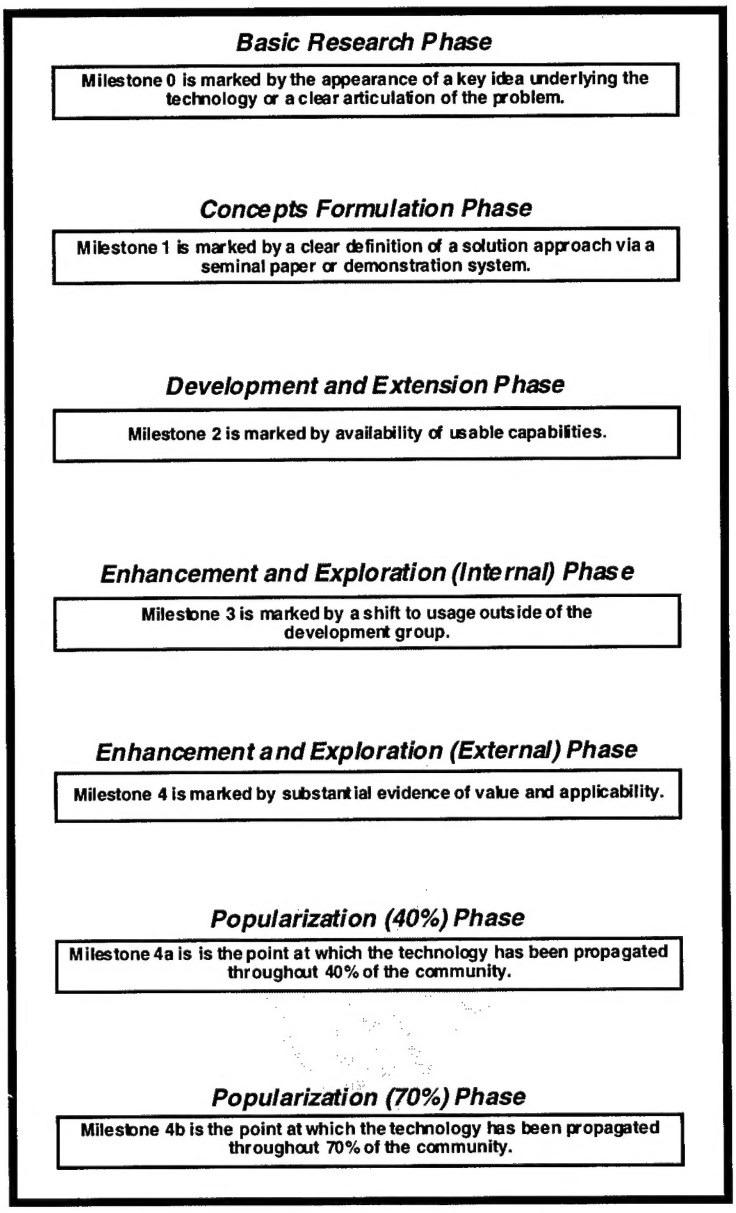


Figure F-8. Phases and milestones for technology maturation [18]

BIBLIOGRAPHY

[1] General Accounting Office, "Software Reuse: Major Issues Need to Be Resolved Before Benefits Can Be Achieved," GAO/IMTEC-93-16, January 1993.

[2] Dunn, Michael F. and John C. Knight, *Certification of Reusable Software Parts*, Department of Computer Science Manuscript, University of Virginia, August 1992.

[3] ANSI/ASQC Q9000-1-1994, American National Standard, Quality Management and Quality Assurance Standards - Guidelines for Selection and Use, American Society for Quality Control, Milwaukee, Wisconsin, August 1, 1994.

[4] Defense Information Systems Agency Center for Software, *Methods of Certifying Non-code Reusable Assets*, Detailed Report, DCA 1000-93-D-0066, December 1994.

[5] Bundy, G.N. W. W. Agresti and W.R. Stewart, *Software Tool Support for Reuse Certification*, The MITRE Corporation, MTR940000109, September 1994.

[6] Software Productivity Solutions, Inc., *Component Certification*, Interim Technical Report for the U.S. Air Force Rome Laboratory, Contract No. F30602-92-C-1058, May 1995.

[7] Software Productivity Solutions, Inc., *Certification of Reusable Software Components (CRC), Volume 1 - Effort Summary*, Draft Interim Technical Report for the U.S. Air Force Rome Laboratory, Contract No. F30602-94-C-0024, April 1996.

[8] Software Productivity Solutions, Inc., *Certification Framework*, Draft Interim Technical Report for the U.S. Air Force Rome Laboratory, Contract No. F30602-94-C-0024, February 1996.

[9] Defense Information Systems Agency Center for Software, DoD Software Reuse Initiative, "Software Reuse Business Model (SRBM)," U.S. Army Space & Strategic Defense Command, Technical Report, January 31, 1995.

[10] U.S. Army Space and Strategic Defense Command Software Engineering Division, "Component Evaluation Procedure (Phase II) Technical Report, January 31, 1995.

[11] National Software Data & Information Repository (NSDIR), "Metrics Collection and Submission Guide, " Volume I - General Instructions, Volume II - Repository Information Request, Volume III - Recurring Data Form, Deputy Assistant Secretary for Communications, Computers, and Logistics, Office of the Assistant Secretary of the Air Force for Acquisition, Version 2.0, 16 June 1995.

[12] Bowen, T.P., et. al., "Specification of Software Quality Attributes." Technical Report RADC-TR-85-37, Rome Laboratory, February 1985.

[13] Software Productivity Solutions, Inc., *Task Area: Software Quality Framework (SQF)*, Interim Technical Report for the U.S. Air Force Rome Laboratory, Contract No., F30602-92-C-0158, October 1995.

[14] Comer, Edward R., et. al., *P1420.1A/D6, Guide for Information Technology - Software Reuse - Asset Certification Framework*, Technical Committee 4: Asset Evaluation and Certification of the Reuse Library Interoperability Group (RIG) January 1996.

[15] Software Productivity Solutions, Inc., *Cost/Benefit Plan*, Draft Interim Technical Report for the U.S. Air Force Rome Laboratory, Contract No. F30602-94-C-0024, February 1996.

[16] Software Productivity Solutions, Inc., *Tool Evaluation for Certification*, Draft Interim Technical Report for the U.S. Air Force Rome Laboratory, Contract No. F30602-94-C-0024, April 1996.

[17] Software Productivity Solutions, Inc., *Certification Field Trials*, Draft Interim Technical Report for the U.S. Air Force Rome Laboratory, Contract No. F30602-94-C-0024, April 1996.

[18] Redwine, S.T. and M.M. Eward, "Software Engineering Technology Transfer Practices," *International Perspectives in Software Engineering*, January 1993, pp. 18-22.

AUTHORS' BIOGRAPHIES

Ms. Sharon L. Rohde is a Sr. Software Engineer at Software Productivity Solutions, Inc., Indialantic, FL. She is a researcher in measurement, reuse and certification and its application to software development.

Ms. Karen A. Dyson is a Sr. Software Engineer at Software Productivity Solutions, Inc., with expertise in developing measurement tools, guidebooks and training to assess the quality of software. She was responsible for the development of the CRC field trials and the analysis of the results.

Dr. Pamela T. Geriner is a Sr. Systems Engineer at Software Productivity Solutions, Inc., FL. Her areas of expertise are strategic planning, economic analyses, business process re-engineering, and total quality management. Dr. Geriner was the technical leader and Program Manager of the CRC effort. Previously, Dr. Geriner was a departmental manager at MITRE's Economic and Decision Analysis Center and a professor at George Mason University.

Ms. Deborah A. Cerino is the Laboratory Project Manager for CRC at RL in Rome, NY. She has many years of experience in software quality, reliability and reuse.

*U.S. GOVERNMENT PRINTING OFFICE: 1997-509-127-47205

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.